VTT Technical Research Centre of Finland

Spine Deliverable 3.1 The Spine Model and its documentation

Ihlemann, Maren; Dillon, Jody; Rasku, Topi; Philipps, Kristof; Marin, Manuel; Mertens, Tim; Huang, Jiangyi; Poncelet, Kris; Kiviluoma, Juha

Published: 30/04/2021

Document Version
Publisher's final version

Spine		
	Co-funded by the European Commission within the H2020 Programme Grant Agreement no: 774629 2017-10-01 until 2021-09-30 (48 months)	

Deliverable 3.1 The Spine Model and its documentation

Revision	1
Preparation date	2021-04-30 (m43)
Due date	2021-04-30 (m43)
Lead contractor	KUL

Author:

Maren Ihlemann	KUL
Jody Dillon	ER
Topi Rasku	VTT
Kristof Philipps	KUL
Manuel Marin	KTH
Tim Mertens	KUL
Jiangyi Huang	VTT
Kris Poncelet	KUL
Juha Kiviluoma	VTT

Dissemination level		
PU	Public	X
CO	Confidential, only for members of the consortium (including the Commission Services)	

Deliverable administration										
No & name	**D3.1 The Spine Model and its documentation**									
Status	Final	**Due**	M43	**Date**	2021-04-30					
Author(s)	Maren Ihlemann KUL, Jody Dillon ER, Topi Rasku VTT, Kristof Philipps KUL, Manuel Marin KTH, Tim Mertens KUL, Jiangyi Huang VTT, Kris Poncelet KUL, Juha Kiviluoma VTT									
Reviewer(s)	Ciara O'Dwyer UCD, Jussi Ikäheimo VTT									
Description of the related task and the deliverable. Extract from DoA	This work package is central to the whole project as it will design and implement the Spine Model that will optimize integrated energy systems (by minimizing the value of the objective function while respecting constraint functions). The work will be carried out in strong collaboration as all partners are familiar with developing energy system models using mathematical programming languages and bring their expertise in specific areas (energy sectors and methodologies) to the table. Task 3.1 designs the core model structures. Task 3.2 implements the data conversion from the Spine Toolbox's generic data structure to the format used by the modelling language. The Spine Model will allow multiple ways to present energy transfers and conversions and will formulate the equations efficiently (Task 3.3). The Spine Model will also enable parallel computing (Task 3.4). Finally, the results will be postprocessed and sent to the Spine Toolbox (Task 3.5). A working model with just core elements will be provided by M11 and a full model by M19. Tasks 3.3 and 3.5 will continue almost through the whole project in order to keep the model updated based on the input from the case studies. D3.1 The Spine Model and its documentation: - A bare bone version of the model that passes data through, M07 - A version for the case studies A1–A5, M10 - A version for the case studies B1–B5, M19 - A version with all the capabilities implemented during the project, M43									
Planned resources PM of WP3	VTT	UCD	KUL	KTH	ER					Total
	13.5	4	15	3.5	5					41.0
Comments										

V	Date	Authors	Description
1.0	2021-04-16	KUL	First draft
1.1	2021-04-23	UCD	Review
1.2	2021-04-30	KUL	Final

Disclaimer

This project has received funding from the European Union's Horizon 2020 research and innovation programme under Grant Agreement No. 774629. ***Topic: LCE-05-2017 Tools and technologies for coordination and integration of the European energy system***

Table of contents

SPINEOPT DOCUMENTATION

The purpose of this document is to give an all-encompassing documentation of the Spine Model. During the project, the name "SpineOpt.jl" was adopted to reflect the Julia package naming convention. This document describes SpineOpt and all its capabilities implemented during the project and applied to the case studies. In line with the ambition to provide open-source and easily accessible software, the documentation of SpineOpt is hosted online and open-source on github. The link to the latest version of the documentation is https://spine-project.github.io/SpineOpt.jl/latest/. The link to the latest version of SpineOpt itself is https://github.com/Spine-project/SpineOpt.jl. For the sake of completeness, the online documentation is attached to the document at hand (accessed: 2021-04-30).

ANNEX: ONLINE DOCUMENTATION OF SPINEOPT

The following pages contain a full copy of the SpineOpt online documentation as they were on 30th of April 2021. The reader is instructed to go to the online version for the latest changes. SpineOpt and its documentation will continue to evolve also after the Spine project ends.

Introduction

SpineOpt.jl is an integrated energy systems optimization model created as part of the Spine project, striving towards adaptability for a multitude of modelling purposes. The data-driven model structure allows for highly customizable energy system descriptions, as well as flexible temporal and stochastic structures, without the need to alter the model source code directly. The methodology is based on mixed-integer linear programming (MILP), and *SpineOpt* relies on *JuMP.jl* for interfacing with the different solvers.

While, in principle, it is possible to run *SpineOpt* by itself, it has been designed to be used through the Spine toolbox, and take maximum advantage of the data and modelling workflow management tools therein. Thus, we highly recommend installing *Spine toolbox* as well, as outlined in the Installation guide.

Contents

In order to make it easier for you to familiarize yourself with the documentation, here's a list of all the different chapters, as well as descriptions of what they're about.

Getting Started

As the name implies, this chapter contains guides for starting to use *SpineOpt.jl* for the first time. The Installation section contains a step-by-step guide for how to install *SpineOpt.jl* and *Spine Toolbox* on your computer. The Setting up a workflow for SpineOpt in Spine Toolbox section explains how to set up and run *SpineOpt.jl* from *Spine Toolbox*. The Creating Your Own Model section explains how to create a new model from scratch. This includes a list of the necessary Object Classes and Relationship Classes, but for more information, you will probably need to consult the **Concept Reference** chapter.

Concept Reference

This chapter lists and explains all the important *data and model structure related concepts* to understand in *SpineOpt.jl*. For a mathematical modelling point of view, see the **Mathematical Formulation** chapter instead. The Basics of the model structure section briefly explains the general purpose of the most important concepts, like Object Classes and Relationship Classes. Meanwhile, the Object Classes, Relationship Classes, Parameters, and Parameter Value Lists sections contain detailed explanations of each and every aspect of *SpineOpt.jl*, organized into the respective sections for clarity.

Mathematical Formulation

This chapter provides the mathematical view of *SpineOpt.jl*, as some of the methodology-related aspects of the model are more easily understood as math than Julia code. The Variables section explains the purpose of each variable in the model, as well as how the variables are related to the different Object Classes and Relationship Classes. The Constraints section contains the mathematical formulation of each constraint, as well as explanations to their purpose and how they are controlled via different Parameters. Finally, the Objective section explains the default objective function used in *SpineOpt.jl*.

Advanced Concepts

This chapter explains some of the more complicated aspects of *SpineOpt.jl* in more detail, hopefully making it easier for you to better understand and apply them in your own modelling. The first few sections focus on aspects of *SpineOpt.jl* that most users are likely to use, or which are more or less required to understand for advanced use. The Temporal Framework section explains how defining *time* works in *SpineOpt.jl*, and how it can be used for different purposes. The Stochastic Framework section details how different stochastic structures can be defined, how they interact with each other, and how this impacts writing Constraints in *SpineOpt.jl*. The Unit commitment section explains how clustered unit-commitment is defined, while the Ramping and Reserves section explains how to enable these operational details in your model. The Investment Optimization section explains how to include investment variables in your models, while the Unit Constraints section details how to include generic data-driven custom constraints.

The last few sections focus on highly specialized use-cases for *SpineOpt.jl*, which are unlikely to be relevant for simple modelling tasks. The Decomposition section explains the Benders decomposition implementation included in *SpineOpt.jl*, as well as how to use it. The remaining sections, namely PTDF-Based Powerflow, Pressure driven gas transfer, Lossless nodal DC power flows, and Representative days with seasonal storages, explain various use-case specific modelling approaches supported by *SpineOpt.jl*.

Installation »

Compatibility

This package requires Julia 1.2 or later.

Installation

SpineOpt is cross-platform (Linux, Mac and Windows) and uses other cross-platform tools. The installation process includes several steps, since there are two other pieces of software that make the use of SpineOpt more convenient (Spine Toolbox and Conda) and two programming languages that are needed (Python for Spine Toolbox and Julia for SpineOpt). Python will be installed with Conda while Julia will be setup for Spine Toolbox (explained below).

You may skip parts of the following installation process if you already have some of these software available - but please make sure they are in a clean Conda environment to avoid compatibility issues between different package versions.

SpineOpt and Spine Toolbox are under active development and the getting started process could change. If you notice any problems with these instructions, please check if it is a known issue, and if not, then report an issue or start a discussion if you're unsure whether it is an actual issue.

- The recommended interface to SpineOpt is Spine Toolbox. Install Spine Toolbox following instructions from here: Spine Toolbox installation
- Setup Julia for Spine Toolbox: Start Spine Toolbox. Go to *File -> Settings -> Tools*. Either select an existing Julia installation or press *Install Julia* and follow the instructions.
- Select a Julia Kernel spec. If there is none, you may need to define and install a Kernel specification using the dialog under *Kernel spec editor*. Use the newly installed Julia, give it a name and a path to a directory where it should put the files related to the Julia project (SpineOpt in this case). The process may also install iJulia, which allows you to interact with Julia code inside Spine Toolbox using the Julia console.
- Install SpineOpt from a Julia console (you can use the Julia console in Spine Toolbox: Go to *Consoles -> Start Julia Console*)

```
julia> using Pkg

julia> pkg"registry add https://github.com/Spine-project/SpineJuliaRegistry"

julia> pkg"add SpineOpt"
```

This may take a while and nothing seems to happen, but the installation process should be ongoing.

- Add SpineOpt tool icons to Spine Toolbox. Go to *Plugins* –> *Install plugins* and select and install SpineOpt.
- You should get a new ribbon in the toolbar with *Run SpineOpt* and *Load template*

After this, you have SpineOpt available as a tool in Spine Toolbox, but next you need to setup a workflow including input and output dabases. Instructions are in the next section here.

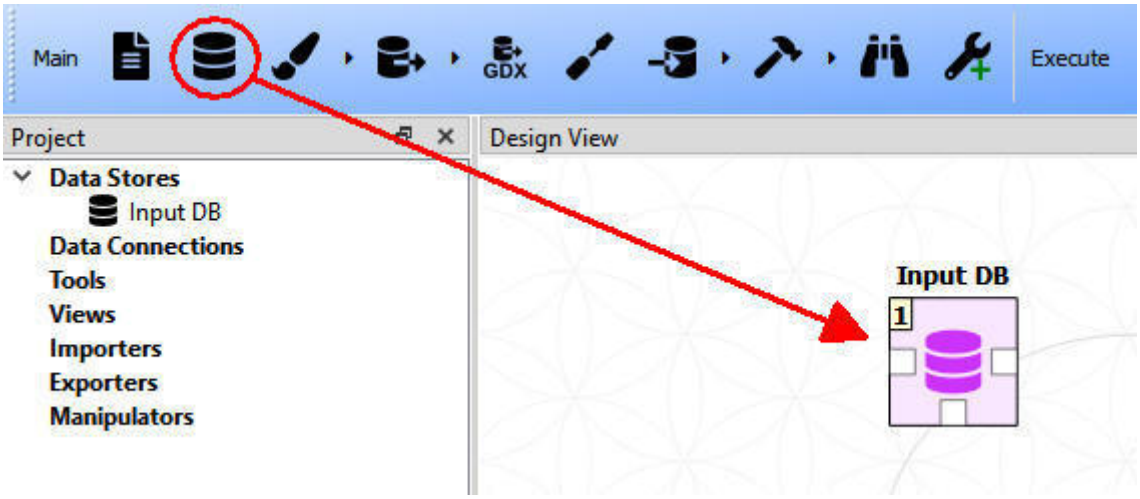
« Introduction

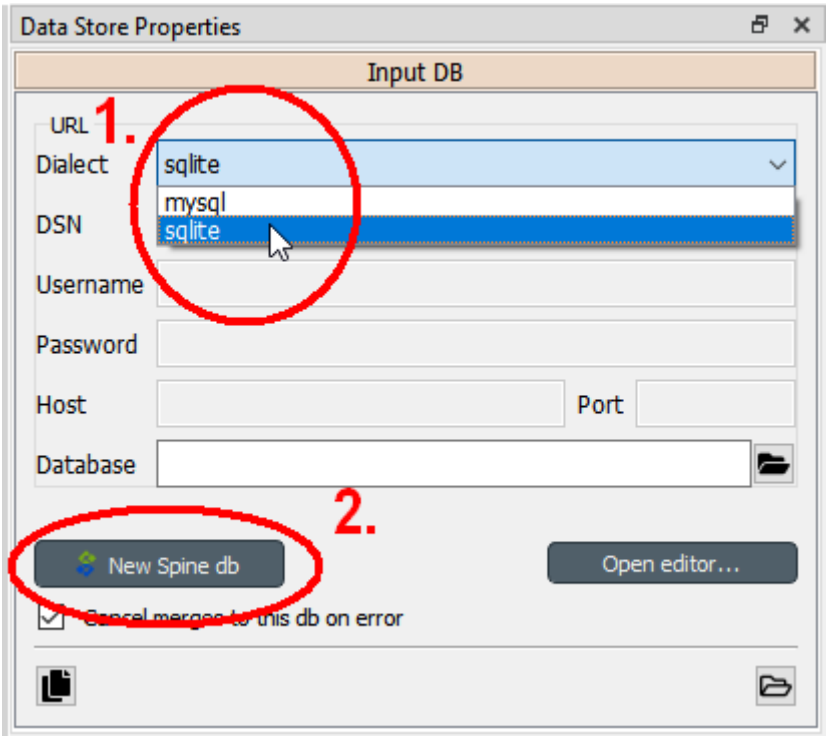
Setting up a workflow »

Setting up a workflow for SpineOpt in Spine Toolbox

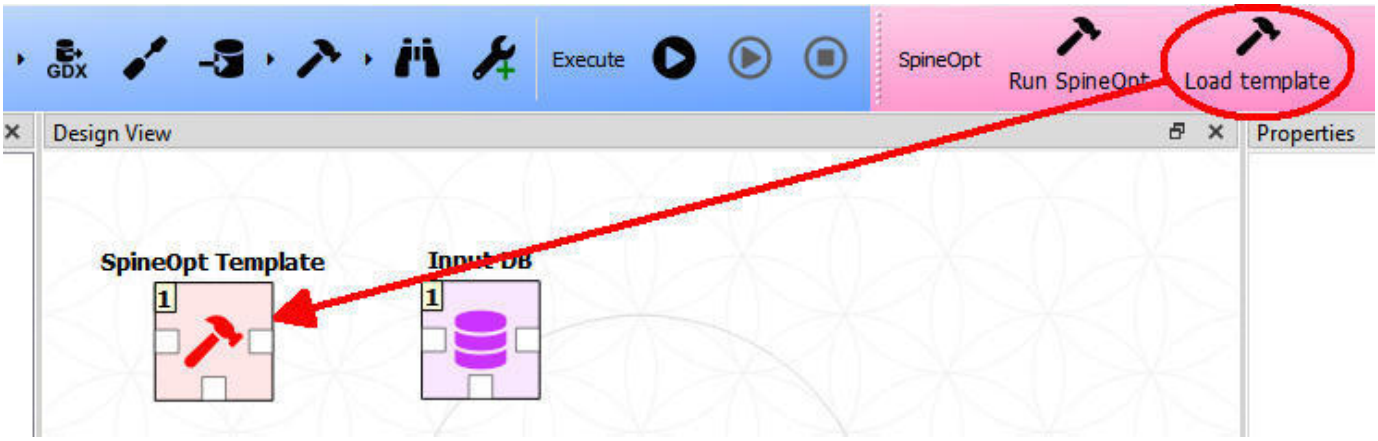
The next steps will set up a SpineOpt specific input database by creating a new Spine database, loading a blank SpineOpt template, connecting it to a SpineOpt instance and setting up a database for model results.

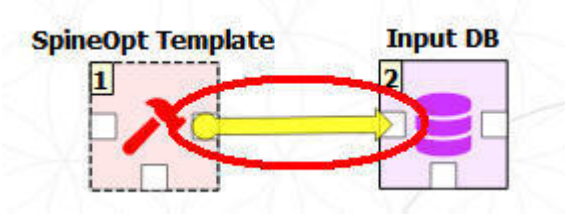
- Create a new Spine Toolbox project in an empty folder of your choice: *File –> New project...*
- Create the input database
    - Drag an empty *Data store* from the toolbar to the *Design View*.
    - Give it a name like "Input DB".
    - Select SQL database dialect (sqlite is a local file and works without a server).
    - Click *New Spine DB* in the *Data Store Properties* window and create a new database (and save it, if it's sqlite).
    - For more information about creating and managing Spine Toolbox database, see the documentation

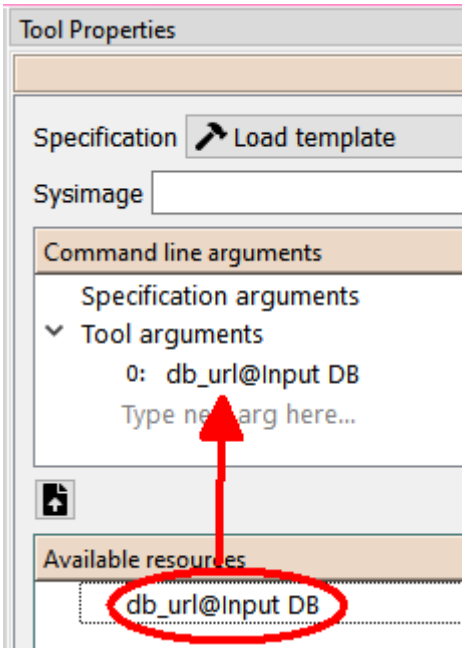


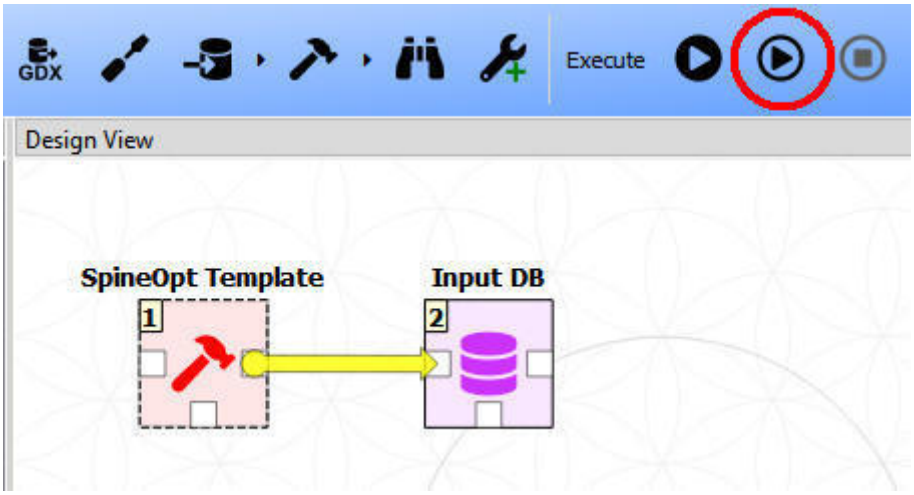


- Fill the *Input DB* with SpineOpt data format **either** by:
    - Drag a tool *Load template* from the SpineOpt ribbon to the *Design View*.
    - Connect an arrow from the *Load template* to the new *Input DB*.
    - Make sure the *Load template* item from the Design view is selected (then you can edit the properties of that workflow item in the *Tool properties* window.
    - Add the url link in *Available resources* to the *Tool arguments* - you are passing the database address as a command line argument to the load_template.jl script so that it knows where to store the output.
    - Then execute the *Load template* tool. Please note that this process uses SpineOpt to generate the data structure. It takes time, since everything is compiled when running a tool in Julia for the first time in each Julia session. You may also see lot of messages and warnings concernging the compilation, but they should be benign.









- ...**or** by:
  - Start Julia (you can start a separate Julia console in Spine Toolbox: go to *Consoles* –> *Start Julia Console*).
  - Copy the URL address of the Data Store from the 'Data Store Properties' –> a copy icon at the bottom.
  - Then run the following script with the right URL address pasted. The process uses SpineOpt itself to build the database structure. Please note that 'using SpineOpt' for the first time for each Julia session takes time - everything is being compiled.

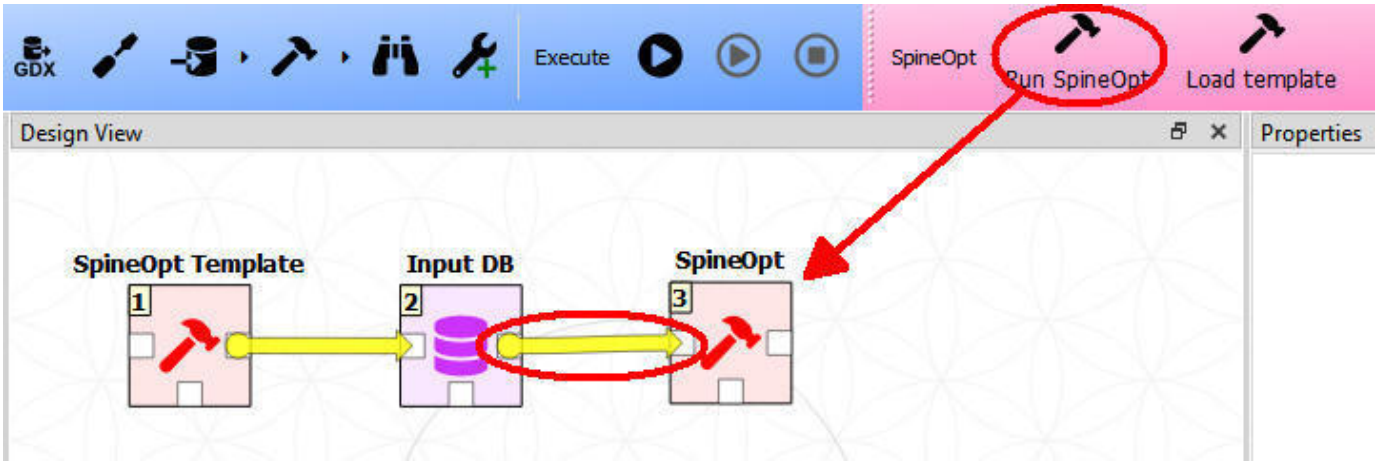
```julia
julia> using SpineOpt

julia> SpineOpt.import_data("copied URL address, inside these quotes", SpineOpt.template(), "Load SpineOpt template")
```

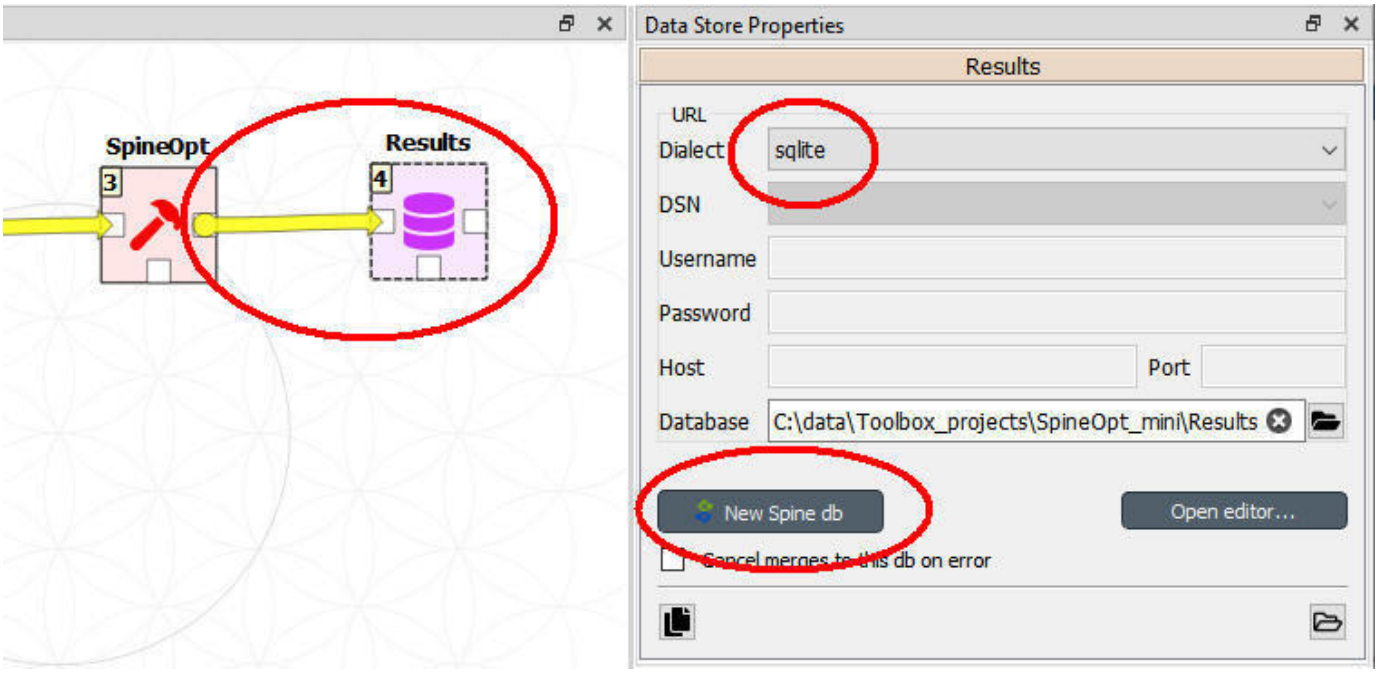
Known issue: On Windows, the backslash between directories need to be

changed to a double forward slash.

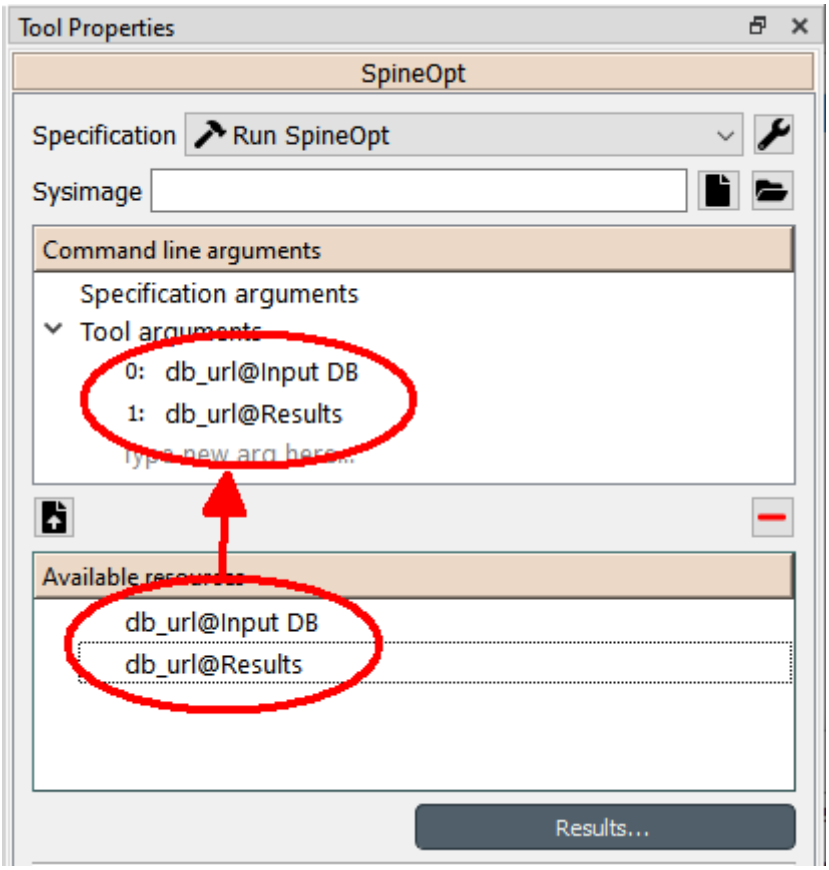
- Drag SpineOpt tool icon to the *Design view*.
- Connect an arrow from the *Input DB* to *SpineOpt*.



- Create a database for results
    - Drag a new *Data store* from the toolbar to the *Design View*.
    - You can rename it to e.g. *Results*. Select SQL database dialect (sqlite is a local file and works without a server).
    - Click *New Spine DB* in the *Data Store Properties* window and create a new database (and save it, if it's sqlite).
    - Connect an arrow from the *SpineOpt* to *Results*.



- Select *SpineOpt* tool in the *Design view*.
- Add the url link for the input data store and the output data store from *Available resources* to the *Tool arguments* (in that order).



SpineOpt would be ready to run, but for the *Input DB*, which is empty of content (it's just a template that contains a SpineOpt specific data structure). The next step goes through setting up and running a simple toy model.

« Installation

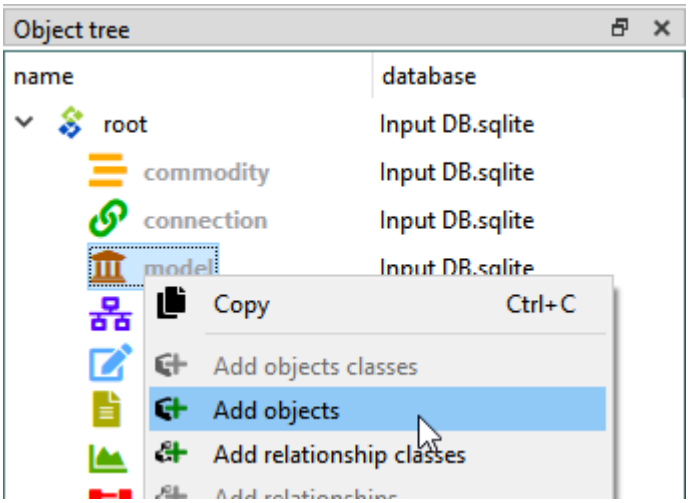
Creating Your Own Model »

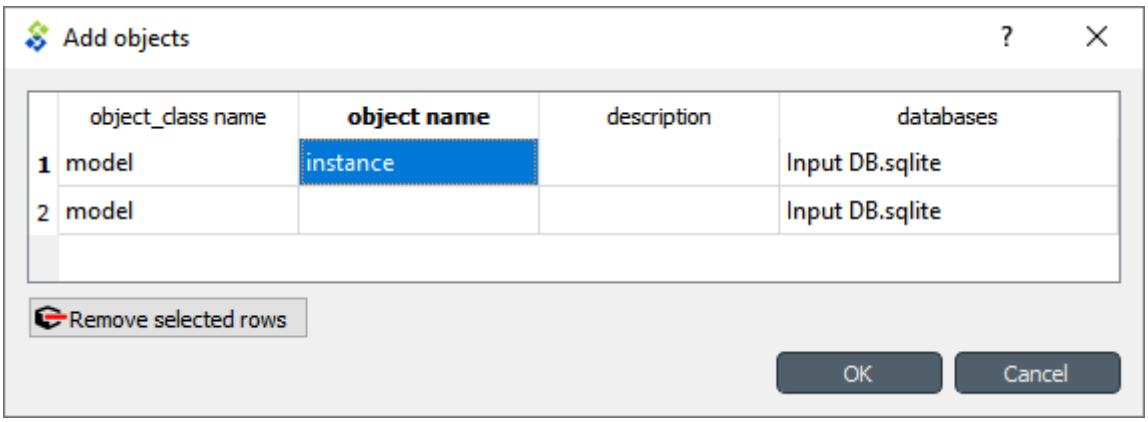
Creating Your Own Model

This part of the guide shows first an example how to insert objects and their parameter data. Then it shows what other objects, relationships and parameter data needs to be added for a very basic model. Lastly, the model instance is run.

Creating a SpineOpt model instance

- First, open the database editor by double-clicking the *Input DB*.
- Right click on *model* in the *Object tree*.
- Choose *Add objects*.
- Then, add a model object by writing a name to the *object name* field. You can use e.g. *instance*.
- Click ok.
- The model object in SpineOpt is an abstraction that represents the model itself. Every SpineOpt database needs to have at least one `model` object.
- The model object holds general information about the optimization. The whole range of functionalities is explained in **Advanced Concepts** chapter - in here a minimal set of parameters is used.





Add parameter values to the model instance

- Select the model object `instance` from the object tree.
- Go to the `Object parameter value` tab.
- Every parameter value belongs to a specific alternative. This allows to hold multiple values for the same parameter of a particular object. The alternative values are used to create scenarios. Choose, `Base` for all parameter values (`Base` is required in Spine Toolbox - all other alternatives can be chosen freely).
- Then define a `model_start` time and a `model_end` time.
    - Double-click on the empty row under `parameter_name` and select model_start.
    - A `None` should appear in `value` column.
    - To asign a start date value, right-click on `None` and open the editor (cannot be entered directly, since the datatype needs to be changed).
    - The parameter type of `model_start` is of type `Datetime`.
    - Set the value to e.g. `2019-01-01T00:00:00`.
    - Proceed accordingly for the model_end.

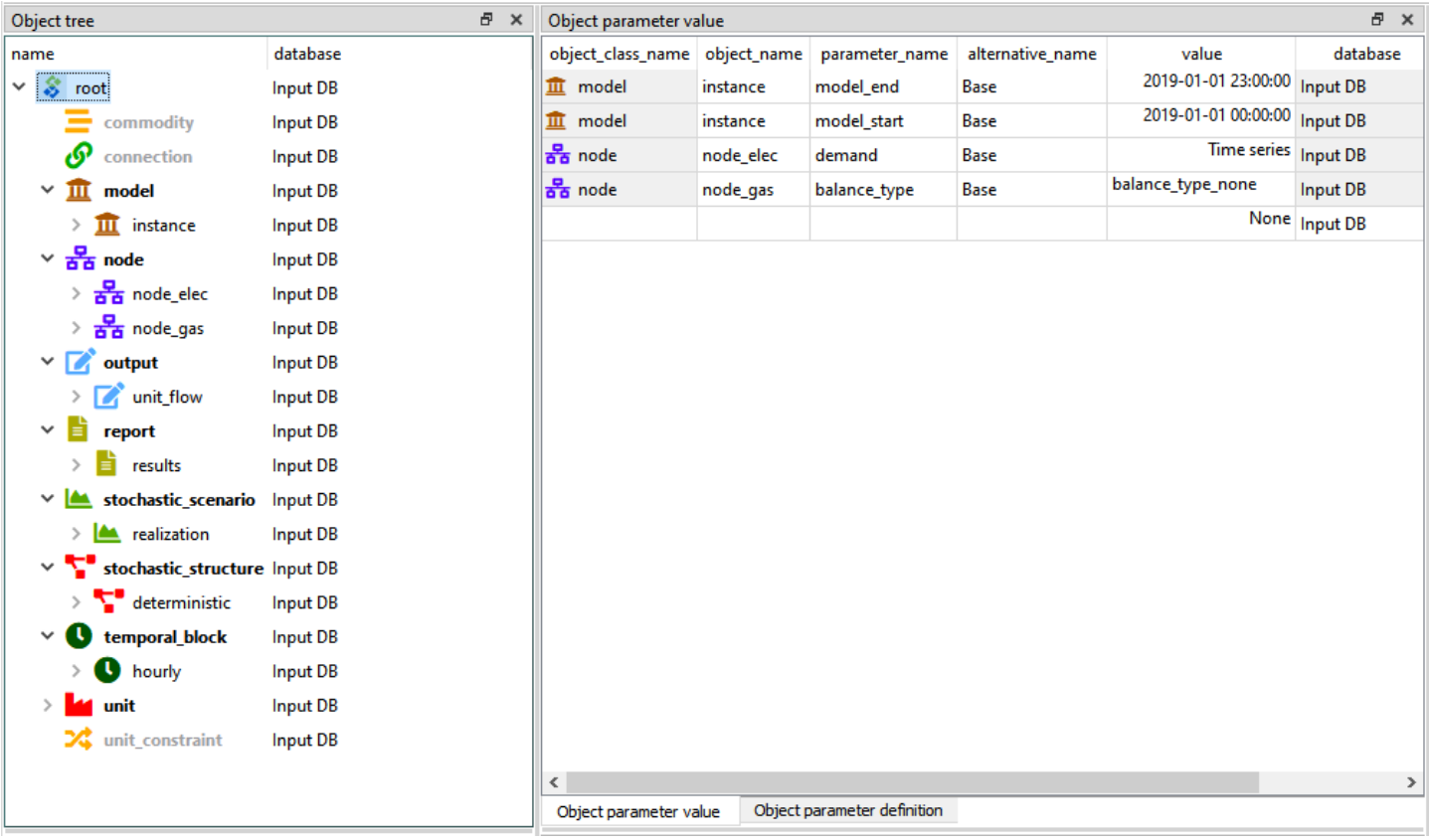
Object parameter value

object_class_name	object_name	parameter_name	alternative_name	value	database
model	instance	model_end	Base	2019-01-01 23:00:00	Input DB
model	instance	model_start	Base	2019-01-01 00:00:00	Input DB
model				None	Input DB

Further reading on adding parameter values can be found here.

Add other necessary objects and parameter data for the objects.

- Add all objects and their parameter data by replicating what has been done in the picture below. Do it the same way as explained above with the following caveats.
- Whilst most object names can be freely defined by the user, there is one object name in the example below that needs to be written exactly since it is used internally by SpineOpt: `unit_flow`.
- The `parameter_name` can be selected from a drop down menu.
- The date time and time series parameter data can be added by using right-click to access the *Edit...* dialog. When creating the time series, use the fixed resolution with `Start time` of the model run and with `1h` resolution. Then only values need to be entered (or copy pasted) and time stamps come automatically.
- Parameter `balance_type` needs to have value `balance_type_none` in the gas node, since it allows the node to create energy (natural gas) against a price and therefore the energy balance is not maintained.



Define temporal and stochastic structures

- To specify the temporal structure for SpineOpt, you need to define temporal_block objects. Think of a `temporal_block` as a distinctive way of 'slicing' time across the model horizon.
- To link the temporal structure to the spatial structure, you need to specify node__temporal_block relationships, establishing which `temporal__block` applies to each `node`. This relationship is added by right-clicking the `node__temporal_block` in the relationship tree and then using the `add relationships...` dialog. Double clicking on an empty cell gives you the list of valid objects. The relationship name is automatically formed, but you can change it if that is desirable.
- To keep things simple at this point, let's just define one `temporal_block` for our model and apply it to all `nodes`. We add the object `hourly_temporal_block` of type `temporal_block` following the same procedure as before and establish `node__temporal_block` relationships between `node_gas` and `hourly_temporal_block`, and `electricity_node` and `hourly_temporal_block`.
- In practical terms, the above means that there energy flows over `gas_node` and `electricity_node` for each 'time-slice' comprised in `hourly_temporal_block`.
- Similarly with the stochastic structure, each node is assigned a `deterministic stochastic_structure`.

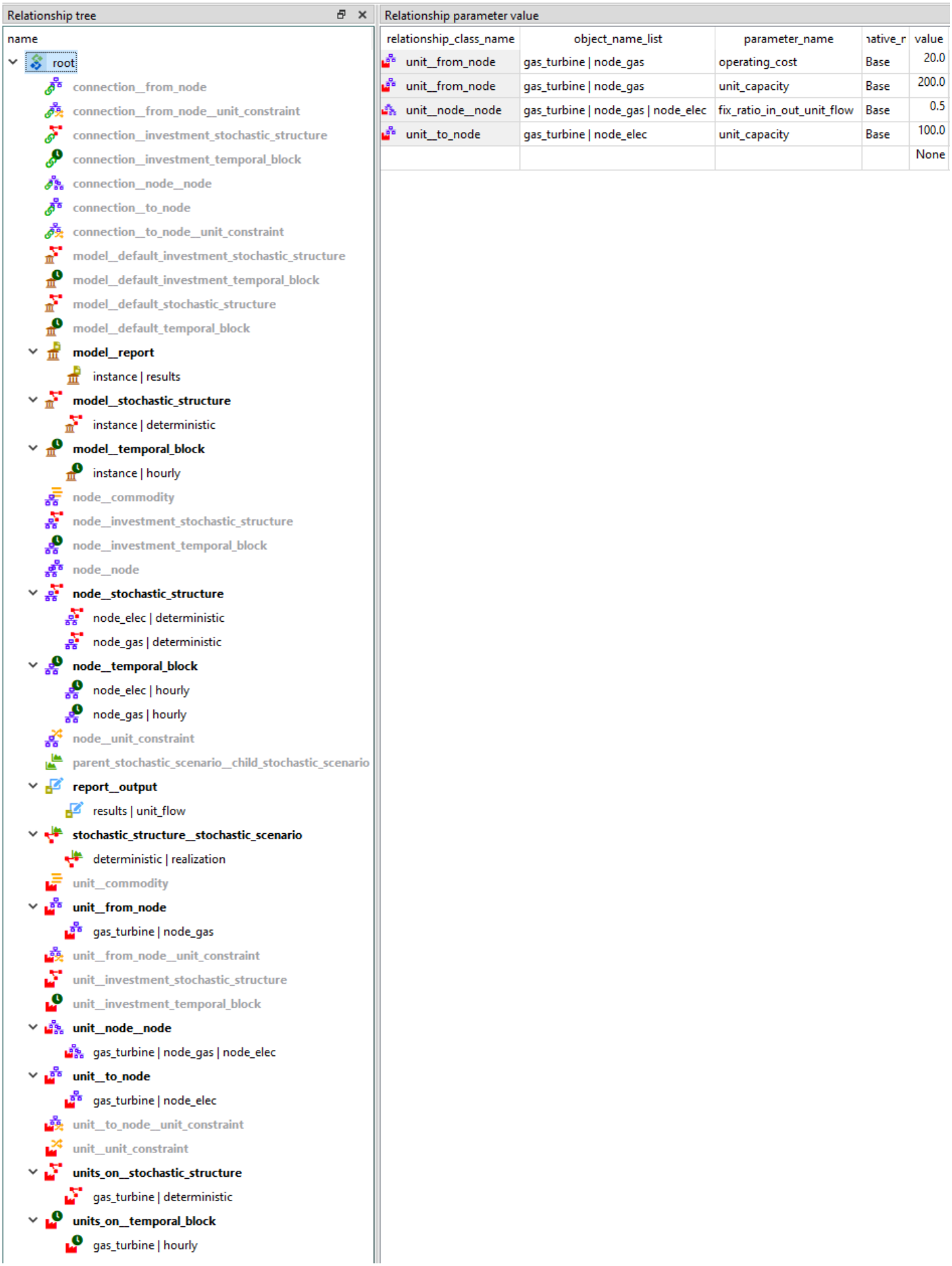
Define the spatial structure

- To specify the spatial structure for SpineOpt, you will need to use the node, unit, and connection objects added before.

- Nodes can be understood as spatial aggregators. In combination with units and connections, they form the energy network.
- Units in SpineOpt represent any kind of conversion process. As one example, a unit can represent a power plant that converts the flow of a commodity fuel into an electricity and/or heat flow.
- Connections on the other hand describe the transport of goods from one location to another. Electricity lines and gas pipelines are examples of such connections. This example does not use connections.
- The database should have an object `gas_turbine` for the `unit` object class and objects `node_gas` and `node_elec` for the `node` object class.
- Next, define how the `unit` and the `nodes` interact with each other: create a unit__from_node relationship between `gas_turbine` and `node_gas`, and unit__to_node relationships between `gas_turbine` and `node_elec`.
- In practical terms, the above means that there is an energy flow going from `node_gas` into `node_elec`, through the `gas_turbine`.

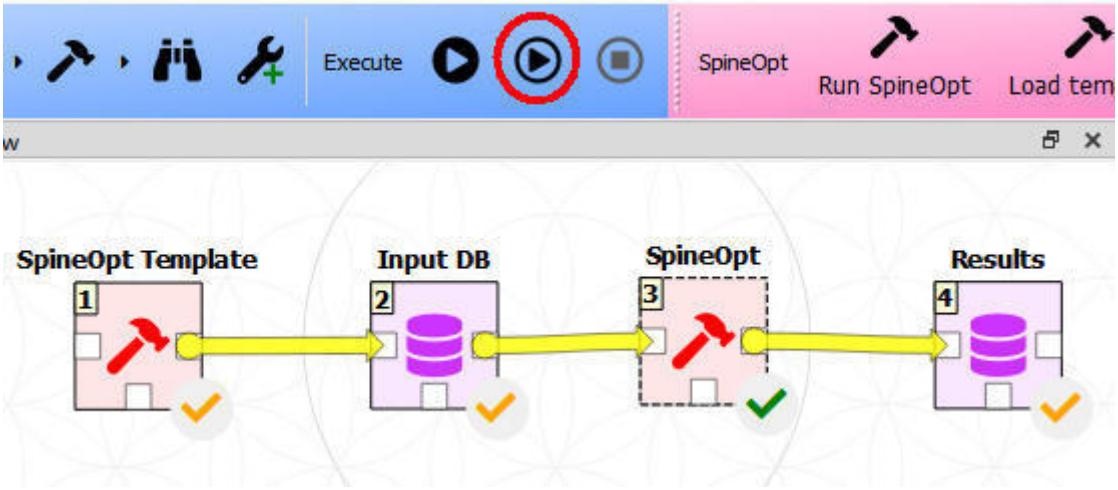
Add remaining relationships and parameter data for the relationships.

- Similar to adding the objects and their parameter data, add the relationships and their parameter data based on the picture below.
- The capacity of the gas_turbine has to be sufficient to meet the highest demand for electricity, otherwise the model will be infeasible (it is possible to set penalty values, but they are not included in this example).
- The parameter `fix_ratio_in_out_unit_flow` forces the ratio between an input and output flow to be a constant. This is one way to establish an efficiency for a conversion process.



Run the model

- Select *SpineOpt*
- Press *Execute selection*.

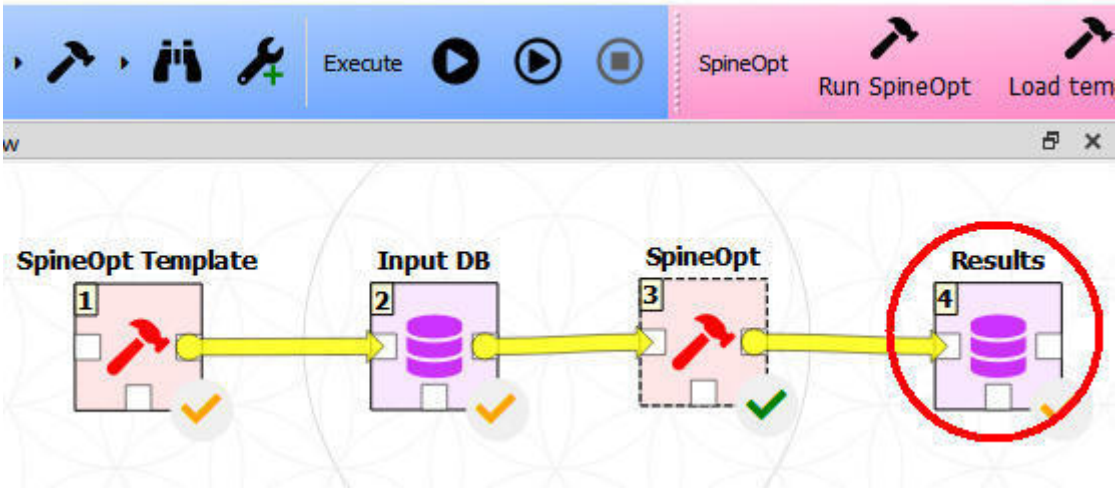


If it fails

- Double-check that the data is correct
- Try to see what the problem might be
- Ask help from the discussion forum

Explore the results

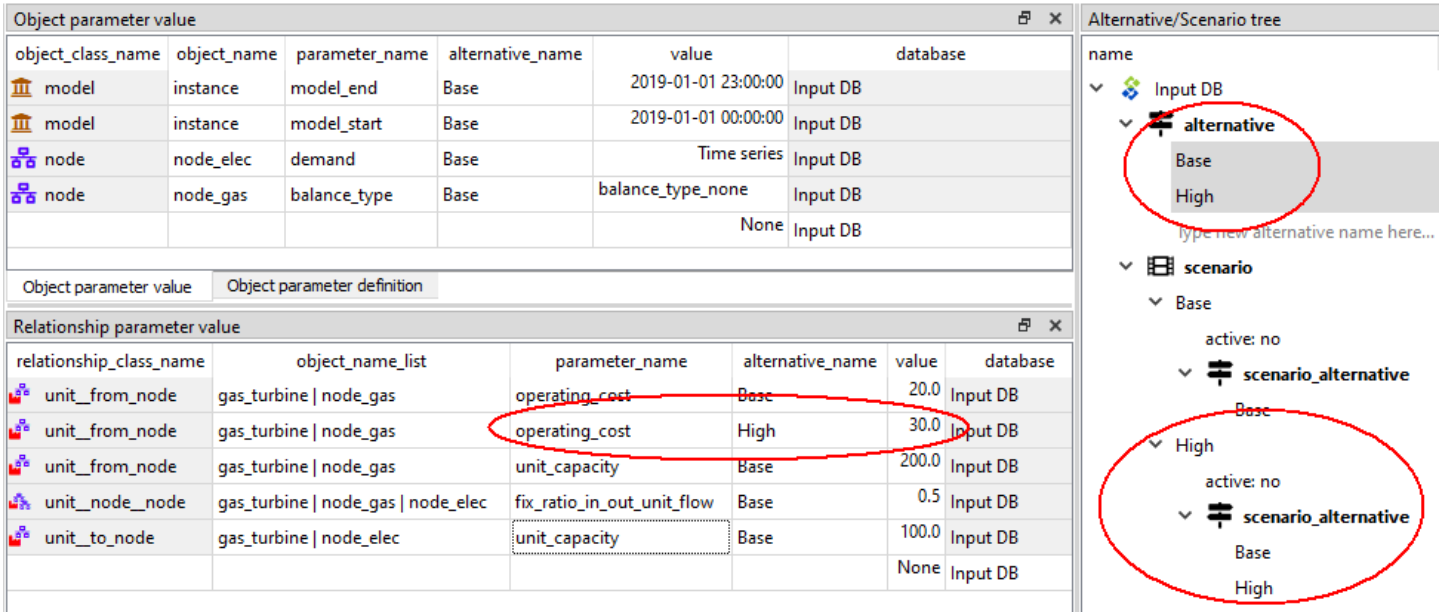
- Double-clicking the *Results* database.

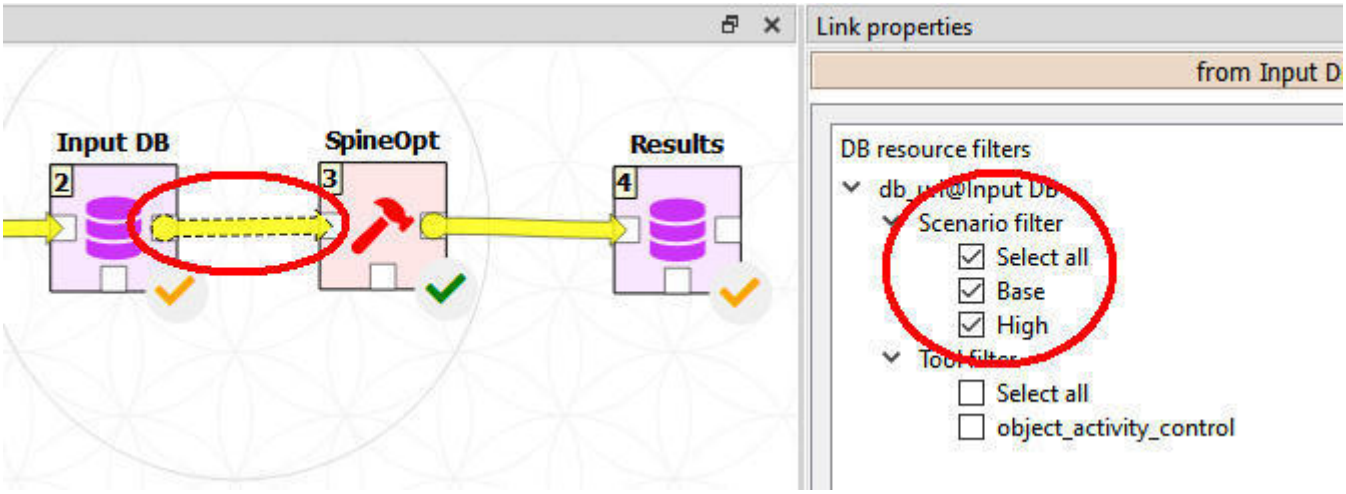


Create and run scenarios and build the model further

- Create a new alternative
- Add parameter data for the new alternative

- Connect alternatives under a scenario. Toolbox modifies `Base` data with the data from the alternatives in the same scenario.
- Execute multiple scenarios in parallel. First run in a new Julia instance will need to compile SpineOpt taking some time.





« Setting up a workflow

Basics of the model structure »

Basics of the model structure

In *SpineOpt.jl*, the model structure is generated based on the input data, allowing it to be used for a multitude of different problems. Here, we aim to provide you with a basic understanding of the *SpineOpt.jl* model and data structure, while the Object Classes, Relationship Classes, Parameters, and Parameter Value Lists sections provide more in-depth explanations of each concept.

Introduction to `object classes`

Essentially, Object Classes represents different types of `objects` or *entities* that make up the model. For example, every power plant in the model is represented as an `object` of the `object class` unit, every power line as an `object` of the `object class` connection, and so forth. In order to add any new *entity* to a model, a new `object` has to be added to desired `object class` in the input data.

Each `object class` has a very specific purpose in *SpineOpt.jl*, so understanding their differences is key. The Object Classes can be roughly divided into three distinctive groups, namely Systemic object classes, Structural object classes, and Meta object classes.

Systemic `object classes`

As the name implies, *system Object Classes* are used to describe the system to be modelled. Essentially, they define *what* you want to model. These include:

- commodity represents different goods to be generated, consumed, transported, etc.
- connection handles the transfer of `commodities` between `nodes`.
- node ensures the balance of the commodity flows, and can be used to store `commodities` as well.
- unit handles the generation and consumption of `commodities`.

Structural `object classes`

Structural Object Classes are used to define the temporal and stochastic structure of the modelled problem, as well as custom Unit Constraints. Unlike the above *system Object Classes*, the *structural Object Classes* are more about *how* you want to model, instead of strictly *what* you want to model. These include:

- stochastic_scenario represents a different *forecast* or another type of an *alternative time period*.
- stochastic_structure acts as a handle for a group of `stochastic_scenarios` with set properties.
- temporal_block defines a period of *time* with the desired temporal resolution.
- unit_constraint is an optional custom constraint generated based on the input data.

Meta `object classes`

Meta Object Classes are used for defining things on the level of `models` or above, like model output and even multiple `models` for problem decompositions. These include:

- model represents an individual *model*, grouping together all the things relevant for itself.
- output defines which Variables are output from the model.
- report groups together multiple output `objects`.

Introduction to `relationship classes`

While Object Classes define all the `objects` or *entities* that make up a model, Relationship Classes define how those *entities* are related to each other. Thus, Relationship Classes hold no meaning on their own, and always include at least one `object class`.

Similar to Object Classes, each `relationship class` has a very specific purpose in *SpineOpt.jl*, and understanding the purpose of each `relationship class` is paramount. The Relationship Classes can be roughly divided into Systemic relationship classes, Structural relationship classes, and Meta relationship classes, again similar to Object Classes.

Systemic `relationship classes`

Systemic Relationship Classes define how Systemic object classes are related to each other, thus helping define the system to be modelled. Most of these relationships deal with *which* `units` and `connections` interact with *which* `nodes`, and *how* those interactions work. This essentially defines the possible commodity flows to be modelled. *Systemic Relationship Classes* include:

- connection__from_node defines which node the connection can transfer a commodity from.
- connection__to_node defines which node the connection can transfer a commodity to.
- connection__node__node holds Parameters for `connections` between two `nodes`.
- node__commodity defines which node holds which commodity.
- node__node holds parameters for direct node-node interactions, like diffusion of `commodities`.
- unit__commodity defines which commodity the unit handles.
- unit__from_node defines which node the unit can take an input commodity from.
- unit__to_node defines which node the unit can output a commodity to.
- unit__node__node holds parameters for unit interactions between two `nodes`.

Structural `relationship classes`

Structural Relationship Classes primarily relate Structural object classes to Systemic object classes, defining what *structures* the individual parts of the *system* use. These are mostly used to determine the

temporal and stochastic structures to be used in different parts of the modelled *system*, or custom Unit Constraints.

SpineOpt.jl has a very flexible temporal and stochastic structure, explained in detail in the Temporal Framework and Stochastic Framework sections of the documentation. Unfortunately, this flexibility requires quite a few different *structural Relationship Classes*, the most important of which are the following *basic structural Relationship Classes*:

- node__stochastic_structure defines the stochastic_structure used for the node balance.
- node__temporal_block defines the `temporal blocks` used for the node balance.
- parent_stochastic_scenario__child_stochastic_scenario defines the *stochastic directed acyclic graph (DAG)* of the Stochastic Framework.
- stochastic_structure__stochastic_scenario holds parameters for `stochastic scenarios` in the stochastic_structure.
- units_on__stochastic_structure defines the stochastic_structure used for the online variable of the unit.
- units_on__temporal_block defines the `temporal blocks` used for the online variable of the unit.

Furthermore, there are also a number of *advanced structural Relationship Classes*, which are only necessary when using some of the optional features of *SpineOpt.jl*. For Investment Optimization, the following relationships control the stochastic and temporal structures of the investment variables:

- connection__investment_stochastic_structure defines the stochastic_structure used for the investment Variables for the connection.
- connection__investment_temporal_block defines the `temporal blocks` used for the investment Variables for the connection.unit_constraint.
- node__investment_stochastic_structure defines the stochastic_structure used for the investment Variables for the node.
- node__investment_temporal_block defines the stochastic_structure used for the investment Variables for the node.
- unit__investment_stochastic_structure defines the stochastic_structure used for the investment Variables for the unit.
- unit__investment_temporal_block defines the `temporal blocks` used for the investment Variables for the unit.(@ref).

For Unit Constraints, which are essentially generic data-driven custom constraints, the following relationships are used to control which variables are included and with what coefficients:

- connection__from_node__unit_constraint holds Parameters for the connection_flow variable *from* the node in question in the custom unit_constraint.
- connection__to_node__unit_constraint holds Parameters for the connection_flow variable *to* the node in question in the custom unit_constraint.
- node__unit_constraint holds Parameters for the node_state variable in the custom unit_constraint.

- unit__from_node__unit_constraint holds Parameters for the unit_flow variable *from* the node in question in the custom unit_constraint.
- unit__to_node__unit_constraint holds Parameters for the unit_flow variable *to* the node in question in the custom unit_constraint.

Meta `relationship classes`

Meta Relationship Classes are used for defining model-level settings, like which `temporal blocks` or `stochastic structures` are active, and what the model output is. These include:

- model__default_investment_stochastic_structure defines a default stochastic_structure to be used for investment Variables when no other definitions exist.
- model__default_investment_temporal_block defines a default temporal_block to be used for investment Variables when no other definitions exist.
- model__default_stochastic_structure defines a default stochastic_structure to be used for `nodes` and `units` when no other definitions exist.
- model__default_temporal_block defines a default temporal_block to be used for `nodes` and `units` when no other definitions exist.
- model__report connects each report to the desired model.
- model__stochastic_structure defines which `stochastic structures` are active in which `models`.
- model__temporal_block defines which `temporal blocks` are active in which `models`.
- report__output defines which `outputs` are part of which report.

Introduction to `parameters`

While the primary function of Object Classes and Relationship Classes is to *define* the system to be modelled and it's structure, Parameters exist to *constrain* them. Every `parameter` is attributed to at least one `object class` or `relationship class`, but some appear in many classes whenever they serve a similar purpose.

Parameters accept different types of values depending on their purpose, e.g. whether they act as a *flag* for some specific functionality or appear as a *coefficient* in Constraints, so understanding each `parameter` is key. Most coefficient-type Parameters accept *constant*, *time series*, and even *stochastic time series* form input, but there are some exceptions. Most *flag-type* Parameters, on the other hand, have a restricted list of acceptable values defined by their Parameter Value Lists.

The existence of some Constraints is controlled based on if the relevant Parameters are defined. As a rule-of-thumb, a `constraint` only gets generated if at least one of the Parameters appearing in it is defined, but one should refer to the appropriate Constraints and Parameters sections when in doubt.

Introduction to groups of objects

Groups of objects are used within SpineOpt for different purposes. To create a group of objects, simply right-click the corresponding Object Class in the *Spine Toolbox* database editor and select `Add object group`. Groups are essentially special `objects`, that act as a single handle for all of its members.

On the one hand, groups can be used in order to impose constraints on the aggregation of a variable, e.g. on the sum of multiple unit_flow variables. Constraints based on parameters associated with the unit__node__node, unit__to_node, unit__from_node, connection__node__node, connection__to_node, connection__from_node can generally be used for this kind of flow aggregation by defining the parameters on groups of objects, typically node groups. (with the exception of variable fixing parameters, e.g. fix_unit_flow, fix_connection_flow etc.). See for instance constraint_unit_flow_capacity.

On the other hand, a node group can be used to for PTDF based powerflows. Here a node group is used to enforce a nodal balance on system level, while suppressing the node balances at individual nodes. See also balance_type and the node balance constraint.

« Creating Your Own Model

Object Classes »

Object Classes

commodity

> A good or product that can be consumed, produced, traded. E.g., electricity, oil, gas, water...

> **Related Parameters:** commodity_lodf_tolerance, commodity_physics, commodity_ptdf_threshold and is_active

> **Related Relationship Classes:** node__commodity and unit__commodity

Commodities correspond to the type of energy traded. When associated with a node through the node__commodity relationship, a specific form of energy, i.e. commodity, can be associated with a specific location. Furthermore, by linking commodities with units, it is possible to track the flows of a certain commodity and impose limitations on the use of a certain commodity (See also max_cum_in_unit_flow_bound). For the representation of specific commodity physics, related to e.g. the representation of the electric network, designated parameters can be defined to enforce commodity specific behaviour. (See also commodity_physics)

connection

> A transfer of commodities between nodes. E.g. electricity line, gas pipeline...

> **Related Parameters:** candidate_connections, connection_availability_factor, connection_contingency, connection_flow_cost, connection_investment_cost, connection_investment_lifetime, connection_investment_variable_type, connection_monitored, connection_reactance_base, connection_reactance, connection_resistance, connection_type, fix_connections_invested_available, fix_connections_invested, has_binary_gas_flow and is_active

> **Related Relationship Classes:** connection__from_node__unit_constraint, connection__from_node, connection__investment_stochastic_structure,

> connection__investment_temporal_block, connection__node__node, connection__to_node__unit_constraint and connection__to_node

A connection represents a transfer of one commodity over space. For example, an electricity transmission line, a gas pipe, a river branch, can be modelled using a connection.

A connection always takes commodities from one or more nodes, and releases them to one or more (possibly the same) nodes. The former are specificed through the connection__from_node relationship, and the latter through connection__to_node. Every connection inherits the temporal and stochastic structures from the associated nodes. The model will generate `connection_flow` variables for every combination of connection, node, *direction* (from node or to node), *time slice*, and *stochastic scenario*, according to the above relationships.

The operation of the connection is specified through a number of parameter values. For example, the capacity of the connection, as the maximum amount of energy that can enter or leave it, is given by connection_capacity. The conversion ratio of input to output can be specified using any of fix_ratio_out_in_connection_flow, max_ratio_out_in_connection_flow, and min_ratio_out_in_connection_flow parameters in the connection__node__node relationship. The delay on a connection, as the time it takes for the energy to go from one end to the other, is given by connection_flow_delay.

`model`

> An instance of SpineOpt, that specifies general parameters such as the temporal horizon.

> **Related Parameters:** big_m, duration_unit, is_active, max_gap, max_iterations, model_end, model_start, model_type, roll_forward, write_lodf_file, write_mps_file and write_ptdf_file

> **Related Relationship Classes:** model__default_investment_stochastic_structure, model__default_investment_temporal_block, model__default_stochastic_structure, model__default_temporal_block, model__report, model__stochastic_structure and model__temporal_block

The model object holds general information about the optimization problem at hand. Firstly, the modelling horizon is specified on the model object, i.e. the scope of the optimization model, and if applicable the duration of the rolling window (see also model_start, model_end and roll_forward). Secondly, the model works as an overarching assembler - only through linking temporal_blocks and stochastic_structures to a model object via relationships, they become part of the optimization problem, and respectively linked nodes, connections and units. If desired the user can also specify

defaults for temporals and stochastic via the designated default relationships (see e.g., model__default_temporal_block). In this case, the default temporal is populated for missing node__temporal_block relationships. Lastly, the model object contains information about the algorithm used for solving the problem (see model_type).

node

> A universal aggregator of commodify flows over units and connections, with storage capabilities.

> **Related Parameters:** balance_type, candidate_storages, demand, downward_reserve, fix_node_pressure, fix_node_state, fix_node_voltage_angle, fix_storages_invested_available, fix_storages_invested, frac_state_loss, fractional_demand, has_pressure, has_state, has_voltage_angle, is_active, is_reserve_node, max_node_pressure, max_voltage_angle, min_node_pressure, min_voltage_angle, minimum_reserve_activation_time, nodal_balance_sense, node_opf_type, node_slack_penalty, node_state_cap, node_state_min, state_coeff, storage_investment_cost, storage_investment_lifetime, storage_investment_variable_type, tax_in_unit_flow, tax_net_unit_flow, tax_out_unit_flow and upward_reserve

> **Related Relationship Classes:** connection__from_node__unit_constraint, connection__from_node, connection__node__node, connection__to_node__unit_constraint, connection__to_node, node__commodity, node__investment_stochastic_structure, node__investment_temporal_block, node__node, node__stochastic_structure, node__temporal_block, node__unit_constraint, unit__from_node__unit_constraint, unit__from_node, unit__node__node, unit__to_node__unit_constraint and unit__to_node

The node is perhaps the most important `object class` out of the Systemic object classes, as it is what connects the rest together via the Systemic relationship classes. Essentially, nodes act as points in the modelled commodity network where commodity balance is enforced via the node balance and node injection constraints, tying together the inputs and outputs from units and connections, as well as any external demand. Furthermore, nodes play a crucial role for defining the temporal and stochastic structures of the model via the node__temporal_block and node__stochastic_structure relationships. For more details about the Temporal Framework and the Stochastic Framework, please refer to the dedicated sections.

Since nodes act as the points where commodity balance is enforced, this also makes them a natural fit for implementing *storage*. The has_state parameter controls whether a node has a `node_state` variable, which essentially represents the commodity content of the node. The state_coeff parameter

tells how the `node_state` variable relates to all the commodity flows. Storage losses are handled via the frac_state_loss parameter, and potential diffusion of commodity content to other nodes via the diff_coeff parameter for the node__node relationship.

`output`

> A variable name from SpineOpt that can be included in a report.

> **Related Parameters:** is_active

> **Related Relationship Classes:** report__output

An output is essentially a handle for a *SpineOpt* variable and Objective function to be included in a report and written into an output database. Typically, e.g. the unit_flow variables are desired as output from most models, so creating an output object called `unit_flow` allows one to designate it as something to be written in the desired report. Note that unless appropriate model__report and report__output relationships are defined, *SpineOpt* doesn't write any output!

`report`

> A results report from a particular SpineOpt run, including the value of specific variables.

> **Related Parameters:** is_active and output_db_url

> **Related Relationship Classes:** model__report and report__output

A report is essentially a group of outputs from a model, that gets written into the output database as a result of running *SpineOpt*. Note that unless appropriate model__report and report__output relationships are defined, *SpineOpt* doesn't write any output!

`stochastic_scenario`

> A scenario for stochastic optimisation in SpineOpt.

> Related **Parameters**: is_active

> Related **Relationship Classes**: parent_stochastic_scenario__child_stochastic_scenario and stochastic_structure__stochastic_scenario

Essentially, a stochastic_scenario is a label for an alternative period of time, describing one possibility of what might come to pass. They are the basic building blocks of the scenario-based Stochastic Framework in *SpineOpt.jl*, but aren't really meaningful on their own. Only when combined into a stochastic_structure using the stochastic_structure__stochastic_scenario and parent_stochastic_scenario__child_stochastic_scenario relationships, along with Parameters like the weight_relative_to_parents and stochastic_scenario_end, they become meaningful.

`stochastic_structure`

> A group of stochastic scenarios that represent a structure.

> Related **Parameters**: is_active

> Related **Relationship Classes**: connection__investment_stochastic_structure, model__default_investment_stochastic_structure, model__default_stochastic_structure, model__stochastic_structure, node__investment_stochastic_structure, node__stochastic_structure, stochastic_structure__stochastic_scenario, unit__investment_stochastic_structure and units_on__stochastic_structure

The stochastic_structure is the key component of the scenario-based Stochastic Framework in *SpineOpt.jl*, and essentially represents a group of stochastic_scenarios with set Parameters. The stochastic_structure__stochastic_scenario relationship defines which stochastic_scenarios are included in which stochastic_structures, and the weight_relative_to_parents and stochastic_scenario_end Parameters define the exact shape and impact of the stochastic_structure, along with the parent_stochastic_scenario__child_stochastic_scenario relationship.

The main reason as to why stochastic_structures are so important is, that they act as handles connecting the Stochastic Framework to the modelled system. This is handled using the Structural relationship classes e.g. node__stochastic_structure, which define the stochastic_structure applied to each `object` describing the modelled system. Connecting each system `object` to the appropriate stochastic_structure individually can be a bit bothersome at times, so there are also a number of

convenience Meta relationship classes like the model__default_stochastic_structure, which allow setting model-wide defaults to be used whenever specific definitions are missing.

temporal_block

> A length of time with a particular resolution.

> **Related Parameters:** block_end, block_start, is_active, representative_periods_mapping, resolution and weight

> **Related Relationship Classes:** connection__investment_temporal_block, model__default_investment_temporal_block, model__default_temporal_block, model__temporal_block, node__investment_temporal_block, node__temporal_block, unit__investment_temporal_block and units_on__temporal_block

A temporal block defines the temporal properties of the optimization that is to be solved in the current window. It is the key building block of the Temporal Framework. Most importantly, it holds the necessary information about the resolution and horizon of the optimization. A single model can have multiple temporal blocks, which is one of the main sources of temporal flexibility in Spine: by linking different parts of the model to different temporal blocks, a single model can contain aspects that are solved with different temporal resolutions or time horizons.

unit

> A conversion of one/many comodities between nodes.

> **Related Parameters:** candidate_units, curtailment_cost, fix_units_invested_available, fix_units_invested, fix_units_on, fom_cost, is_active, min_down_time, min_up_time, number_of_units, online_variable_type, shut_down_cost, start_up_cost, unit_availability_factor, unit_investment_cost, unit_investment_lifetime and unit_investment_variable_type

> **Related Relationship Classes:** unit__commodity, unit__from_node__unit_constraint, unit__from_node, unit__investment_stochastic_structure, unit__investment_temporal_block, unit__node__node, unit__to_node__unit_constraint, unit__to_node, unit__unit_constraint, units_on__stochastic_structure and units_on__temporal_block

A unit represents an energy conversion process, where energy of one commodity can be converted into energy of another commodity. For example, a gas turbine, a power plant, or even a load, can be modelled using a unit.

A unit always takes energy from one or more nodes, and releases energy to one or more (possibly the same) nodes. The former are specificed through the unit__from_node relationship, and the latter through unit__to_node. Every unit has a temporal and stochastic structures given by the units_on__temporal_block and [units_on__stochastic_structure] relationships. The model will generate `unit_flow` variables for every combination of unit, node, *direction* (from node or to node), *time slice*, and *stochastic scenario*, according to the above relationships.

The operation of the unit is specified through a number of parameter values. For example, the capacity of the unit, as the maximum amount of energy that can enter or leave it, is given by unit_capacity. The conversion ratio of input to output can be specified using any of fix_ratio_out_in_unit_flow, max_ratio_out_in_unit_flow, and min_ratio_out_in_unit_flow. The variable operating cost is given by vom_cost.

unit_constraint

> A generic data-driven custom constraint.

> **Related Parameters:** constraint_sense, is_active and right_hand_side

> **Related Relationship Classes:** connection__from_node__unit_constraint, connection__to_node__unit_constraint, node__unit_constraint, unit__from_node__unit_constraint, unit__to_node__unit_constraint and unit__unit_constraint

The unit_constraint is a generic data-driven custom constraint, which allows for defining constraints involving multiple units, nodes, or connections. The constraint_sense parameter changes the sense of the unit_constraint, while the right_hand_side parameter allows for defining the constant terms of the constraint.

Coefficients for the different variables appearing in the unit_constraint are defined using relationships, like e.g. unit__from_node__unit_constraint and connection__to_node__unit_constraint for unit_flow and connection_flow variables, or unit__unit_constraint and node__unit_constraint for units_on, units_started_up, and node_state variables.

For more information, see the dedicated article on Unit Constraints

---

« Basics of the model structure

Relationship Classes »

Relationship Classes

connection__from_node

> Defines the nodes the connection can take input from, and holds most connection_flow variable specific parameters.

> **Related Object Classes:** connection and node

> **Related Parameters:** connection_capacity, connection_conv_cap_to_flow, connection_emergency_capacity, fix_binary_gas_connection_flow, fix_connection_flow, fix_connection_intact_flow and graph_view_position

int

connection__from_node is a two-dimensional relationship between a connection and a node and implies a connection_flow to the connection from the node. Specifying such a relationship will give rise to a connection_flow_variable with indices connection=connection, node=node, direction=:from_node. Relationships defined on this relationship will generally apply to this specific flow variable. For example, connection_capacity will apply only to this specific flow variable, unless the connection parameter connection_type is specified.

c_structure

block

connection__from_node__unit_constraint

> when specified this relationship allows the relevant flow connection flow variable to be included in the specified user constraint

> **Related Object Classes:** connection, node and unit_constraint

re

> **Related Parameters:** connection_flow_coefficient

connection__from_node__unit_constraint is a three-dimensional relationship between a connection, a node and a unit_constraint. The relationship specifies that the connection_flow variable

to the specified connection from the specified node is involved in the specified unit_constraint. Parameters on this relationship generally apply to this specific `connection_flow` variable. For example the parameter connection_flow_coefficient defined on `connection__from_node__unit_constraint` represents the coefficient on the specific `connection_flow` variable in the specified unit_constraint

connection__investment_stochastic_structure

> Defines the stochastic structure of the connections investments variable

> **Related Object Classes:** connection and stochastic_structure

The connection__investment_stochastic_structure relationship defines the stochastic_structure of connection-related investment decisions. Essentially, it sets the stochastic_structure used by the connections_invested_available variable of the connection.

The connection__investment_stochastic_structure relationship uses the model__default_investment_stochastic_structure relationship if not defined.

connection__investment_temporal_block

> Defines the temporal resolution of the connections investments variable

> **Related Object Classes:** connection and temporal_block

`connection__investment_temporal_block` is a two-dimensional relationship between a connection and a temporal_block. This relationship defines the temporal resolution and scope of a connection's investment decision. Note that in a decomposed investments problem with two model objects, one for the master problem model and another for the operations problem model, the link to the specific model is made indirectly through the model__temporal_block relationship. If a model__default_investment_temporal_block is specified and no `connection__investment_temporal_block` relationship is specified, the model__default_investment_temporal_block relationship will be used. Conversely if `connection__investment_temporal_block` is specified along with model__temporal_block, this will override model__default_investment_temporal_block for the specified connection.

See also Investment Optimization

tochastic_scenario

connection__node__node

enario

> Holds parameters spanning multiple `connection_flow` variables to and from multiple `nodes`.

re

> **Related Object Classes:** connection and node

> **Related Parameters:** compression_factor, connection_flow_delay, connection_linepack_constant, fix_ratio_out_in_connection_flow, fixed_pressure_constant_0, fixed_pressure_constant_1, max_ratio_out_in_connection_flow and min_ratio_out_in_connection_flow

`connection__node__node` is a three-dimensional relationship between a connection, a node (node 1) and another node (node 2). `connection__node__node` infers a conversion and a direction with respect to that conversion. Node 1 is assumed to be the input node and node 2 is assumed to be the output node. For example, the fix_ratio_out_in_connection_flow parameter defined on `connection__node__node` relates the output `connection_flow` to node 2 to the intput `connection_flow` from node 1

in

tructure

ok

connection__to_node

t

> Defines the `nodes` the `connection` can output to, and holds most `connection_flow` variable specific parameters.

c_structure

block

> **Related Object Classes:** connection and node

> **Related Parameters:** connection_capacity, connection_conv_cap_to_flow, connection_emergency_capacity, fix_connection_flow, fix_connection_intact_flow and graph_view_position

re

`connection__to_node` is a two-dimensional relationship between a connection and a node and implies a `connection_flow` from the connection to the node. Specifying such a relationship will give rise to a `connection_flow_variable` with indices `connection=connection, node=node, direction=:to_node`. Relationships defined on this relationship will generally apply to this specific flow variable. For example, connection_capacity will apply only to this specific flow variable, unless the connection parameter connection_type is specified.

tochastic_scenario

connection__to_node__unit_constraint

enario

when specified this relationship allows the relevant flow connection flow variable to be included in the specified user constraint

re

Related Object Classes: connection, node and unit_constraint

Related Parameters: connection_flow_coefficient

`connection__to_node__unit_constraint` is a three-dimensional relationship between a connection, a node and a unit_constraint. The relationship specifies that the `connection_flow` variable from the specified connection to the specified node is involved in the specified unit_constraint. Parameters on this relationship generally apply to this specific `connection_flow` variable. For example the parameter connection_flow_coefficient defined on `connection__to_node__unit_constraint` represents the coefficient on the specific `connection_flow` variable in the specified unit_constraint

int

model__default_investment_stochastic_structure

tructure

ck

Defines the default stochastic structure used for investment variables, which will be replaced by more specific definitions

t

c_structure

Related Object Classes: model and stochastic_structure

block

The model__default_investment_stochastic_structure relationship can be used to set model-wide default unit__investment_stochastic_structure, connection__investment_stochastic_structure, and node__investment_stochastic_structure relationships. Its main purpose is to allow users to avoid defining each relationship individually, and instead allow them to focus on defining only the exceptions. As such, any specific unit__investment_stochastic_structure, connection__investment_stochastic_structure, and node__investment_stochastic_structure relationships take priority over the model__default_investment_stochastic_structure relationship.

re

model__default_investment_temporal_block

Defines the default temporal block used for investment variables, which will be replaced by more specific definitions

tochastic_scenario

enario

> **Related Object Classes:** model and temporal_block

`model__default_investment_temporal_block` is a two-dimensional relationship between a model and a temporal_block. This relationship defines the default temporal resolution and scope for all investment decisions in the model (units, connections and storages). Specifying `model__default_investment_temporal_block` for a model avoids the need to specify individual node__investment_temporal_block, unit__investment_temporal_block and connection__investment_temporal_block relationships. Conversely, if any of these individual relationships are defined (e.g. connection__investment_temporal_block) along with model__temporal_block, these will override model__default_investment_temporal_block.

re

See also Investment Optimization

model__default_stochastic_structure

> Defines the default stochastic structure used for model variables, which will be replaced by more specific definitions

int

tructure

ck

> **Related Object Classes:** model and stochastic_structure

The model__default_stochastic_structure relationship can be used to set a model-wide default for the node__stochastic_structure and units_on__stochastic_structure relationships. Its main purpose is to allow users to avoid defining each relationship individually, and instead allow them to focus on defining only the exceptions. As such, any specific node__stochastic_structure or units_on__stochastic_structure relationships take priority over the model__default_stochastic_structure relationship.

t

e_structure

block

model__default_temporal_block

> Defines the default temporal block used for model variables, which will be replaced by more specific definitions

re

> **Related Object Classes:** model and temporal_block

The model__default_temporal_block relationship can be used to set a model-wide default for the node__temporal_block and units_on__temporal_block relationships. Its main purpose is to allow users to

tochastic_scenario
avoid defining each relationship individually, and instead allow them to focus on defining only the exceptions. As such, any specific node__temporal_block or units_on__temporal_block relationships take priority over the model__default_temporal_block relationship.
enario

model__report

re

> Determines which reports are written for each model and in turn, which outputs are written for each model

> **Related Object Classes:** model and report

The model__report relationship tells which reports are written by which model, where the contents of the reports are defined separately using the report__output relationship. Without appropriately defined model__report and report__output and relationships, *SpineOpt* doesn't write any output, so be sure to include at least one report connected to all the output variables of interest in the model!

model__stochastic_structure

int
tructure
ck

> Defines which `stochastic_structures` are included in which `models`.

t

> **Related Object Classes:** model and stochastic_structure

c_structure
The [model__stochastic_structure] relationship defines which stochastic_structures are active in which models. Essentially, this relationship allows for e.g. attributing multiple node__stochastic_structure relationships for a single node, and switching between them in different models. Any stochastic_structure in the model__default_stochastic_structure relationship is automatically assumed to be active in the connected model, so there's no need to include it in [model__stochastic_structure] separately.
block

model__temporal_block

re

> Defines which `temporal_blocks` are included in which `models`.

> **Related Object Classes:** model and temporal_block

The `model__temporal_block` relationship is used to determine which temporal_blocks are included in a specific model. Note that defining this relationship does not yet imply that any element of the model will be governed by the specified `temporal_block`, for this to happen additional relationships have to be defined such as the model__default_temporal_block relationship.

node__commodity

Define a `commodity` for a `node`. Only a single `commodity` is permitted per `node`

Related Object Classes: commodity and node

`node__commodity` is a two-dimensional relationship between a node and a commodity and specifies the commodity that `flows` to or from the node. Generally, since flows are not dimensioned by commodity, this has no meaning in terms of the variables and constraint equations. However, there are two specific uses for this relationship:

1. To specify that specific network physics should apply to the network formed by the member nodes for that commodity. See powerflow
2. Only connection flows that are between nodes of the same or no commodity are included in the `node_balance` constraint.

node__investment_stochastic_structure

defines the stochastic structure for node related investments, currently only storages

Related Object Classes: node and stochastic_structure

The node__investment_stochastic_structure relationship defines the stochastic_structure of node-related investment decisions. Essentially, it sets the stochastic_structure used by the storages_invested_available variable of the node.

The node__investment_stochastic_structure relationship uses the model__default_investment_stochastic_structure relationship if not defined.

node__investment_temporal_block

defines the temporal resolution for node related investments, currently only storages

tochastic_scenario

enario

Related Object Classes: node and temporal_block

node__investment_temporal_block is a two-dimensional relationship between a node and a temporal_block. This relationship defines the temporal resolution and scope of a node's investment de decisions (currently only storage invesments). Note that in a decomposed investments problem with two model objects, one for the master problem model and another for the operations problem model, the link to the specific model is made indirectly through the model__temporal_block relationship. If a model__default_investment_temporal_block is specified and no node__investment_temporal_block relationship is specified, the model__default_investment_temporal_block relationship will be used. Conversely if node__investment_temporal_block is specified along with model__temporal_block, this will override model__default_investment_temporal_block for the specified node.

See also Investment Optimization

node__node

int

Holds parameters for direct interactions between two nodes, e.g. node_state diffusion coefficients.

tructure

ck

Related Object Classes: node

t

c_structure

Related Parameters: diff_coeff

block

The node__node relationship is used for defining direct interactions between two nodes, like diffusion of commodity content. Note that the node__node relationship is assumed to be one-directional, meaning that

```
node__node(node1=n1, node2=n2) != node__node(node1=n2, node2=n1).
```

Thus, when one wants to define *symmetric relationships* between two nodes, one needs to define both re directions as separate relationships.

node__stochastic_structure

tochastic_scenario

Defines which specific `stochastic_structure` is used by the `node` and all `flow` variables associated with it. Only one `stochastic_structure` is permitted per `node`.

enario

Related Object Classes: node and stochastic_structure

re

The node__stochastic_structure relationship defines which stochastic_structure the node uses. Essentially, it sets the stochastic_structure of all the `flow` variables connected to the node, as well as the potential node_state variable. Note that only one stochastic_structure can be defined per node per model, as interpreted based on the node__stochastic_structure and model__stochastic_structure relationships. Investment variables use dedicated relationships, as detailed in the Investment Optimization section.

The node__stochastic_structure relationship uses the model__default_stochastic_structure relationship if not specified.

node__temporal_block

int

Defines the `temporal_blocks` used by the `node` and all the `flow` variables associated with it.

tructure

ck

Related Object Classes: node and temporal_block

t

c_structure

Related Parameters: cyclic_condition

block

This relationship links a node to a temporal_block and as such it will determine which temporal block governs the temporal horizon and resolution of the variables associated with this node. Specifically, the resolution of the temporal block will directly imply the duration of the time slices for which both the regular and ramping flow variables and their associated constraints are created.

For a more detailed description of how the temporal structure in SpineOpt can be created, see Temporal Framework.

re

node__unit_constraint

specifying this relationship allows a node's demand or node_state to be included in the specified unit constraint

tochastic_scenario

> Related Object Classes: node and unit_constraint

enario

> Related Parameters: demand_coefficient and node_state_coefficient

node__unit_constraint is a two-dimensional relationship between a node and a unit_constraint. The relationship specifies that a variable associated only with the node (currently only the node_state) is involved in the constraint. For example, the node_state_coefficient defined on node__unit_constraint specifies the coefficient of the node's node_state variable in the specified unit_constraint.

See also unit_constraint

parent_stochastic_scenario__child_stochastic_scenario

> Defines the master stochastic direct acyclic graph, meaning how the stochastic_scenarios are related to each other.

int

tructure

ck

> Related Object Classes: stochastic_scenario

The parent_stochastic_scenario__child_stochastic_scenario relationship defines how the individual stochastic_scenarios are related to each other, forming what is referred to as the *stochastic direct acyclic graph (DAG)* in the Stochastic Framework section. It acts as a sort of basis for the stochastic_structures, but doesn't contain any Parameters necessary for describing how it relates to the Temporal Framework or the Objective function.

t

c_structure

block

The parent_stochastic_scenario__child_stochastic_scenario relationship and the *stochastic DAG* it forms are crucial for Constraint generation with stochastic path indexing. Every finite *stochastic DAG* has a limited number of unique ways of traversing it, called *full stochastic paths*, which are used when determining how many different constraints need to be generated over time periods where stochastic_structures branch or converge, or when generating constraints involving different stochastic_structures. See the Stochastic Framework section for more information.

re

report__output

> Output object related to a report object are returned to the output database (if they appear in the model as variables)

tochastic_scenario

> Related Object Classes: output and report

enario

The report__output relationship tells which output variables to include in which report when writing *SpineOpt* output. Note that the reports also need to be connected to a model using the model__report relationship. Without appropriately defined model__report and report__output and relationships, *SpineOpt* doesn't write any output, so be sure to include at least one report connected to all the output variables of interest in the model!
re

stochastic_structure__stochastic_scenario

> Defines which `stochastic_scenarios` are included in which `stochastic_structure`, and holds the parameters required for realizing the structure in combination with the `temporal_blocks`.

> Related Object Classes: stochastic_scenario and stochastic_structure

int

> Related Parameters: stochastic_scenario_end and weight_relative_to_parents

tructure

ck

The stochastic_structure__stochastic_scenario relationship defines which stochastic_scenarios are included in which stochastic_structure, as well as holds the stochastic_scenario_end and weight_relative_to_parents Parameters defining how the stochastic_structure interacts with the Temporal Framework and the Objective function. Along with parent_stochastic_scenario__child_stochastic_scenario, this relationship is used to define the exact properties of each stochastic_structure, which are then applied to the `objects` describing the modelled system according to the Structural relationship classes, like the node__stochastic_structure relationship.
t

c_structure

block

unit__commodity

> Holds parameters for `commodities` used by the `unit`.
re

> Related Object Classes: commodity and unit

> Related Parameters: max_cum_in_unit_flow_bound

tochastic_scenario

To impose a limit on the cumulative amount of commodity flows, the max_cum_in_unit_flow_bound can
enario be imposed on a unit__commodity relationship. This can be very helpful, e.g. if a certain amount of emissions should not be surpased throughout the optimization.

Note that, next to the unit__commodity relationship, also the nodes connected to the units need to be associated with their corresponding commodities, see node__commodity.
re

unit__from_node

> Defines the `nodes` the `unit` can take input from, and holds most `unit_flow` variable specific parameters.

> **Related Object Classes:** node and unit

> **Related Parameters:** fix_nonspin_ramp_up_unit_flow, fix_nonspin_units_started_up, fix_ramp_up_unit_flow, fix_start_up_unit_flow, fix_unit_flow_op, fix_unit_flow, fuel_cost, graph_view_position, max_res_shutdown_ramp, max_res_startup_ramp, max_shutdown_ramp, max_startup_ramp, min_res_shutdown_ramp, min_res_startup_ramp, min_shutdown_ramp, min_startup_ramp, minimum_operating_point, operating_points, ramp_down_cost, ramp_down_limit, ramp_up_cost, ramp_up_limit, reserve_procurement_cost, unit_capacity, unit_conv_cap_to_flow and vom_cost

int
tructure
ck
t
e_structure

The unit__to_node and unit__from_node unit relationships are core elements of SpineOpt. For each
block unit__to_node or unit__from_node, a unit_flow variable is automatically added to the model, i.e. a commodity flow of a unit *to* or *from* a specific node, respectively.

Various parameters can be defined on the unit__from_node relationship, in order to constrain the associated unit flows. In most cases a unit_capacity will be defined for an upper bound on the commodity flows. Apart from that, ramping abilities of a unit can be defined. For further details on ramps see Ramping and Reserves.

To associate costs with a certain commodity flows, cost terms, such as fuel_costs and vom_costs, can be
re included for the unit__from_node relationship.

It is important to note, that the parameters associated with the unit__from_node can be defined either for a specific node, or for a group of nodes. Grouping nodes for the described parameters will result in an aggregation of the unit flows for the triggered constraint, e.g. the definition of the unit_capacity on a group of nodes will result in an upper bound on the sum of all individual unit_flows.

tochastic_scenario

unit__from_node__unit_constraint

enario

Defines which input `unit_flows` are included in the `unit_constraint`, and holds their parameters.

re

Related Object Classes: node, unit_constraint and unit

Related Parameters: graph_view_position and unit_flow_coefficient

`unit__from_node__unit_constraint` is a three-dimensional relationship between a unit, a node and a unit_constraint. The relationship specifies that the `unit_flow` variable to the specified unit from the specified node is involved in the specified unit_constraint. Parameters on this relationship generally apply to this specific `unit_flow` variable. For example the parameter unit_flow_coefficient defined on `unit__from_node__unit_constraint` represents the coefficient on the specific `unit_flow` variable in the specified unit_constraint

int

unit__investment_stochastic_structure

tructure

ck

Sets the stochastic structure for investment decisions - overrides `model__default_investment_stochastic_structure`.

t

c_structure

Related Object Classes: stochastic_structure and unit

block

The unit__investment_stochastic_structure relationship defines the stochastic_structure of unit-related investment decisions. Essentially, it sets the stochastic_structure used by the units_invested_available variable of the unit.

The unit__investment_stochastic_structure relationship uses the model__default_investment_stochastic_structure relationship if not defined.

re

unit__investment_temporal_block

Sets the temporal resolution of investment decisions - overrides `model__default_investment_temporal_block`

tochastic_scenario

Related Object Classes: temporal_block and unit

enario

`unit__investment_temporal_block` is a two-dimensional relationship between a unit and a temporal_block. This relationship defines the temporal resolution and scope of a unit's investment decision. Note that in a decomposed investments problem with two model objects, one for the master problem model and another for the operations problem model, the link to the specific model is made indirectly through the model__temporal_block relationship. If a model__default_investment_temporal_block is specified and no `unit__investment_temporal_block` relationship is specified, the model__default_investment_temporal_block relationship will be used. Conversely if `unit__investment_temporal_block` is specified along with model__temporal_block, this will override model__default_investment_temporal_block for the specified unit.

See also Investment Optimization

unit__node__node

Holds parameters spanning multiple `unit_flow` variables to and from multiple `nodes`.

int

truct

ck

Related Object Classes: node and unit

t

c_structure

block

Related Parameters: fix_ratio_in_in_unit_flow, fix_ratio_in_out_unit_flow, fix_ratio_out_in_unit_flow, fix_ratio_out_out_unit_flow, fix_units_on_coefficient_in_in, fix_units_on_coefficient_in_out, fix_units_on_coefficient_out_in, fix_units_on_coefficient_out_out, max_ratio_in_in_unit_flow, max_ratio_in_out_unit_flow, max_ratio_out_in_unit_flow, max_ratio_out_out_unit_flow, max_units_on_coefficient_in_in, max_units_on_coefficient_in_out, max_units_on_coefficient_out_in, max_units_on_coefficient_out_out, min_ratio_in_in_unit_flow, min_ratio_in_out_unit_flow, min_ratio_out_in_unit_flow, min_ratio_out_out_unit_flow, min_units_on_coefficient_in_in, min_units_on_coefficient_in_out, min_units_on_coefficient_out_in, min_units_on_coefficient_out_out, unit_idle_heat_rate, unit_incremental_heat_rate and unit_start_flow

re

While the relationships unit__to_node and unit__to_node take care of the automatic generation of the unit_flow variables, the unit__node__node relationships hold the information how the different commodity flows of a unit interact. Only through this relationship and the associated parameters, the topology of a unit, i.e. which intakes lead to which products etc., becomes unambiguous.

In almost all cases, at least one of the `..._ratio_...` parameters will be defined, e.g. to set a fixed ratio between outgoing and incoming commodity flows of unit (see also e.g. fix_ratio_out_in_unit_flow). Note that the parameters can also be defined on a relationship between groups of objects, e.g. to force a fixed

ratio between a group of nodes. In the triggered constraints, this will lead to an aggregation of the individual unit flows.

unit__to_node

> Defines the `nodes` the `unit` can output to, and holds most `unit_flow` variable specific parameters.

> **Related Object Classes:** node and unit

> **Related Parameters:** fix_nonspin_ramp_down_unit_flow, fix_nonspin_ramp_up_unit_flow, fix_nonspin_units_shut_down, fix_nonspin_units_started_up, fix_ramp_down_unit_flow, fix_ramp_up_unit_flow, fix_shut_down_unit_flow, fix_start_up_unit_flow, fix_unit_flow_op, fix_unit_flow, fuel_cost, graph_view_position, max_res_shutdown_ramp, max_res_startup_ramp, max_shutdown_ramp, max_startup_ramp, min_res_shutdown_ramp, min_res_startup_ramp, min_shutdown_ramp, min_startup_ramp, minimum_operating_point, operating_points, ramp_down_cost, ramp_down_limit, ramp_up_cost, ramp_up_limit, reserve_procurement_cost, unit_capacity, unit_conv_cap_to_flow and vom_cost

The unit__to_node and unit__from_node unit relationships are core elements of SpineOpt. For each unit__to_node or unit__from_node, a unit_flow variable is automatically added to the model, i.e. a commodity flow of a unit *to* or *from* a specific node, respectively.

Various parameters can be defined on the unit__to_node relationship, in order to constrain the associated unit flows. In most cases a unit_capacity will be defined for an upper bound on the commodity flows. Apart from that, ramping abilities of a unit can be defined. For further details on ramps see Ramping and Reserves.

To associate costs with a certain commodity flow, cost terms, such as fuel_costs and vom_costs, can be included for the unit__to_node relationship.

It is important to note, that the parameters associated with the unit__to_node can be defined either for a specific node, or for a group of nodes. Grouping nodes for the described parameters will result in an aggregation of the unit flows for the triggered constraint, e.g. the definition of the unit_capacity on a group of nodes will result in an upper bound on the sum of all individual unit_flows.

unit__to_node__unit_constraint

tochastic_scenario

Defines which output `unit_flows` are included in the `unit_constraint`, and holds their parameters.

enario

Related Object Classes: node, unit_constraint and unit

re

Related Parameters: graph_view_position and unit_flow_coefficient

`unit__to_node__unit_constraint` is a three-dimensional relationship between a unit, a node and a unit_constraint. The relationship specifies that the `unit_flow` variable from the specified unit to the specified node is involved in the specified unit_constraint. Parameters on this relationship generally apply to this specific `unit_flow` variable. For example the parameter unit_flow_coefficient defined on `unit__to_node__unit_constraint` represents the coefficient on the specific `unit_flow` variable in the specified unit_constraint

unit__unit_constraint

int

tructure

ck

Defines which `units_on` variables are included in the `unit_constraint`, and holds their parameters.

t

Related Object Classes: unit_constraint and unit

c_structure

block

Related Parameters: units_on_coefficient and units_started_up_coefficient

`unit__unit_constraint` is a two-dimensional relationship between a unit and a unit_constraint. The relationship specifies that a variable or variable(s) associated only with the unit (not a `unit_flow` for example) are involved in the constraint. For example, the units_on_coefficient defined on `unit__unit_constraint` specifies the coefficient of the unit's `units_on` variable in the specified unit_constraint.

re

See also unit_constraint

units_on__stochastic_structure

tochastic_scenario

Defines which specific `stochastic_structure` is used for the `units_on` variable of the `unit`. Only one `stochastic_structure` is permitted per `unit`.

enario

Related Object Classes: stochastic_structure and unit

re

The units_on__stochastic_structure relationship defines the stochastic_structure used by the units_on variable. Essentially, this relationship permits defining a different stochastic_structure for the online decisions regarding the units_on variable, than what is used for the production unit_flow variables. A common use-case is e.g. using only one units_on variable across multiple stochastic_scenarios for the unit_flow variables. Note that only one units_on__stochastic_structure relationship can be defined per unit per model, as interpreted by the units_on__stochastic_structure and model__stochastic_structure relationships.

The units_on__stochastic_structure relationship uses the model__default_stochastic_structure relationship if not specified.

units_on__temporal_block

int

tructure

Defines which specific `temporal_blocks` are used by the `units_on` variable of the `unit`.

ck

Related Object Classes: temporal_block and unit

t

c_structure

units_on__temporal_block is a relationship linking the units_on variable of a unit to a specific temporal_block object. As such, this relationship will determine which temporal block governs the on- and offline status of the unit. The temporal block holds information on the temporal scope and resolution for which the variable should be optimized.

re

Parameters

balance_type

> A selector for how the `:nodal_balance` constraint should be handled.

> **Default value:** balance_type_node

> **Uses Parameter Value Lists:** balance_type_list

> **Related Object Classes:** node

The balance_type parameter determines whether or not a node needs to be balanced, in the classical sense that the sum of flows entering the node is equal to the sum of flows leaving it.

The values `balance_type_node` (the default) and `balance_type_group` mean that the node is always balanced. The only exception is if the node belongs in a group that has itself balance_type equal to `balance_type_group`. The value `balance_type_none` means that the node doesn't need to be balanced.

big_m

> Sufficiently large number used for linearization bilinear terms, e.g. to enforce bidirectional flow for gas pipielines

> **Default value:** 1000000

> **Related Object Classes:** model

The big_m parameter is a property of the model object. The bigM method is commonly used for the purpose of recasting non-linear constraints into a mixed-integer reformulation. In SpineOpt, the bigM formulation is used to describe the sign of gas flow through a connection (if a pressure driven gas transfer model is used). The big_m parameter in combination with the binary variable binary_gas_connection_flow is used in the constraints on the gas flow capacity and the fixed node pressure points and ensures that the average flow through a pipeline is only in one direction and is constraint by the fixed pressure points from the outer approximation of the Weymouth equation. See Schwele - Coordination of Power and Natural Gas Systems: Convexification Approaches for Linepack Modeling for reference.

`block_end`

> The end time for the `temporal_block`. Can be given either as a `DateTime` for a static end point, or as a `Duration` for an end point relative to the start of the current optimization.

> **Related Object Classes:** temporal_block

Indicates the end of this temporal block. The default value is equal to a duration of 0. It is useful to distinguish here between two cases: a single solve, or a rolling window optimization.

single solve When a Date time value is chosen, this is directly the end of the optimization for this temporal block. In a single solve optimization, a combination of block_start and block_end can easily be used to run optimizations that cover only part of the model horizon. Multiple temporal_block objects can then be used to create optimizations for disconnected time periods, which is commonly used in the method of representative days. The default value coincides with the model_end.

rolling window optimization To create a temporal block that is rolling along with the optimization window, a rolling temporal block, a duration value should be chosen. The block_end parameter will in this case determine the size of the optimization window, with respect to the start of each optimization window. If multiple temporal blocks with different block_end parameters exist, the maximum value will determine the size of the optimization window. Note, this is different from the roll_forward parameter, which determines how much the window moves for after each optimization. For more info, see One single `temporal_block`. The default value is equal to the roll_forward parameter.

`block_start`

> The start time for the `temporal_block`. Can be given either as a `DateTime` for a static start point, or as a `Duration` for an start point relative to the start of the current optimization.

> **Related Object Classes:** temporal_block

Indicates the start of this temporal block. The main use of this parameter is to create an offset from the model start. The default value is equal to a duration of 0. It is useful to distinguish here between two cases: a single solve, or a rolling window optimization.

single solve When a Date time value is chosen, this is directly the start of the optimization for this temporal block. When a duration is chosen, it is added to the model_start to obtain the start of this temporal_block. In the case of a duration, the chosen value directly marks the offset of the optimization with respect to the model_start. The default value for this parameter is the model_start.

rolling window optimization To create a temporal block that is rolling along with the optimization window, a rolling temporal block, a duration value should be chosen. The temporal block_start will again mark the offset of the optimization start but now with respect to the start of each optimization window.

`candidate_connections`

> The number of connections that may be invested in

> **Related Object Classes:** connection

The candidate_connections parameter denotes the possibility of investing on a certain connection.

The default value of `nothing` means that the connection can't be invested in, because it's already in operation. An integer value represents the maximum investment possible at any point in time, as a factor of the connection_capacity.

In other words, candidate_connections is the upper bound of the connections_invested_available variable.

`candidate_storages`

> Determines the maximum number of new storages which may be invested in

> **Related Object Classes:** node

Within an investments problem `candidate_storages` determines the upper bound on the storages investment decision variable in constraint `storages_invested_available`. In constraint

`node_state_cap` the maximum node state will be the product of the storages investment variable and node_state_cap. Thus, the interpretation of `candidate_storages` depends on storage_investment_variable_type which determines the investment decision variable type. If storage_investment_variable_type is integer or binary, then `candidate_storages` represents the maximum number of discrete storages of size node_state_cap that may be invested in at the corresponding node. If storage_investment_variable_type is continuous, `candidate_storages` is more analagous to a maximum storage capacity with `node_state_cap` being analagous to a scaling parameter.

Note that `candidate_storages` is the main investment switch and setting a value other than none/nothing triggers the creation of the investment variable for storages at the corresponding node. Note that a value of zero will still trigger the variable creation but its value will be fixed to zero. This can be useful if an inspection of the related dual variables will yield the value of this resource.

See also Investment Optimization and storage_investment_variable_type

`candidate_units`

> Number of units which may be additionally constructed

> **Related Object Classes:** unit

Within an investments problem `candidate_units` determines the upper bound on the unit investment decision variable in constraint `units_invested_available`. In constraint `unit_flow_capacity` the maximum `unit_flow` will be the product of the `units_invested_available` and the corresponding unit_capacity. Thus, the interpretation of `candidate_units` depends on unit_investment_variable_type which determines the unit investment decision variable type. If unit_investment_variable_type is integer or binary, then `candidate_units` represents the maximum number of discrete units that may be invested in. If unit_investment_variable_type is continuous, `candidate_units` is more analagous to a maximum storage capacity.

Note that `candidate_units` is the main investment switch and setting a value other than none/nothing triggers the creation of the investment variable for the unit. Note that a value of zero will still trigger the variable creation but its value will be fixed to zero. This can be useful if an inspection of the related dual variables will yield the value of this resource.

See also Investment Optimization and unit_investment_variable_type

`commodity_lodf_tolerance`

> The minimum absolute value of the line outage distribution factor (LODF) that is considered meaningful.

> **Default value:** 0.1

> **Related Object Classes:** commodity

Given two connections, the line outage distribution factor (LODF) is the fraction of the pre-contingency flow on the first one, that will flow on the second after the contingency. commodity_lodf_tolerance is the minimum absolute value of the LODF that is considered meaningful. Any value below this tolerance (in absolute value) will be treated as zero.

The LODFs are used to model contingencies on some connections and their impact on some other connections. To model contingencies on a connection, set connection_contingency to `true`; to study the impact of such contingencies on another connection, set connection_monitored to `true`.

In addition, define a commodity with commodity_physics set to commodity_physics_lodf, and associate that commodity (via node__commodity) to both connections' nodes (given by connection__to_node and connection__from_node).

commodity_physics

> Defines if the `commodity` follows lodf or ptdf physics.

> **Default value:** commodity_physics_none

> **Uses Parameter Value Lists:** commodity_physics_list

> **Related Object Classes:** commodity

This parameter determines the specific formulation used to carry out dc load flow within a model. To enable power transfer distribution factor (ptdf) based load flow for a network of nodes and connections, all nodes must be related to a commodity with commodity_physics set to commodity_physics_ptdf. To enable security constraint unit comment based on ptdfs and line outage

distribution factors (lodf) all nodes must be related to a commodity with commodity_physics set to commodity_physics_lodf.

See also powerflow

commodity_ptdf_threshold

> The minimum absolute value of the power transfer distribution factor (PTDF) that is considered meaningful.

> **Default value:** 0.0001

> **Related Object Classes:** commodity

Given a connection and a node, the power transfer distribution factor (PTDF) is the fraction of the flow injected into the node that will flow on the connection. commodity_ptdf_threshold is the minimum absolute value of the PTDF that is considered meaningful. Any value below this threshold (in absolute value) will be treated as zero.

The PTDFs are used to model DC power flow on certain connections. To model DC power flow on a connection, set connection_monitored to `true`.

In addition, define a commodity with commodity_physics set to either commodity_physics_ptdf, or commodity_physics_lodf. and associate that commodity (via node__commodity) to both connections' nodes (given by connection__to_node and connection__from_node).

compression_factor

> The compression factor establishes a compression from an origin node to a receiving node, which are connected through a connection. The first node corresponds to the origin node, the second to the (compressed) destination node. Typically the value is >=1.

> **Related Relationship Classes:** connection__node__node

This parameter is specific to the use of pressure driven gas transfer. To represent a compression between two nodes in the gas network, the compression_factor can be defined. This factor ensures that the pressure of a node is equal to (or lower than) the pressure at the sending node times the

compression_factor. The relationship connection__node__node that hosts this parameter should be defined in a way that the first node represents the origin node and the second node represents the compressed node.

connection_availability_factor

> Availability of the `connection`, acting as a multiplier on its `connection_capacity`. Typically between 0-1.

> **Default value:** 1.0

> **Related Object Classes:** connection

To indicate that a connection is only available to a certain extent or at certain times of the optimization, the connection_availability_factor can be used. A typical use case could be an availability timeseries for connection with expected outage times. By default the availability factor is set to `1`. The availability is, among others, used in the constraint_connection_flow_capacity.

connection_capacity

> Limits the `connection_flow` variable to the `to_node`. `to_node` can be a group of `nodes`, in which case the sum of the `connection_flow` is constrained.

> **Related Relationship Classes:** connection__from_node and connection__to_node

Defines the upper bound on the corresponding `connection_flow` variable. If the connection is a candidate connection, the effective `connection_flow` upper bound is the product of the investment variable, `connections_invested_available` and `connection_capacity`. If ptdf based dc load flow is enabled, `connection_capacity` represents the normal rating of a `connection` (line) while connection_emergency_capacity represents the maximum post contingency flow.

connection_contingency

> A boolean flag for defining a contingency `connection`.

> **Uses Parameter Value Lists:** boolean_value_list

> **Related Object Classes:** connection

Specifies that the connection in question is to be included as a contingency when security constrained unit commitment is enabled. When using security constrained unit commitment by setting commodity_physics to commodity_physics_lodf, an N-1 security constraint is created for each monitored line (`connection_monitored` = `true`) for each specified contingency (`connection_contingency` = `true`).

See also powerflow

connection_conv_cap_to_flow

> Optional coefficient for `connection_capacity` unit conversions in the case the `connection_capacity` value is incompatible with the desired `connection_flow` units.

> **Default value:** 1.0

> **Related Relationship Classes:** connection__from_node and connection__to_node

The connection_conv_cap_to_flow can be used to perform the conversion between the measurement unit of the connection_capacity to the measurement unit of the connection_flow variable. The default of this parameter is `1`, i.e. assuming that both are given in the same measurement unit.

connection_emergency_capacity

> The maximum post-contingency flow on a monitored `connection`.

> **Related Relationship Classes:** connection__from_node and connection__to_node

The connection_emergency_capacity parameter represents the maximum post-contingency flow on a *monitored* connection if ptdf and lodf based security constrained unit commitment is enabled (commodity_physics is set to [commodity_physics_lodf]).

If you set this value, make sure that you also set connection_monitored to `true` for the involved connection.

connection_flow_coefficient

> defines the unit constraint coefficient on the connection flow variable in the to direction

> **Default value:** 0.0

> **Related Relationship Classes:** connection__from_node__unit_constraint and connection__to_node__unit_constraint

The connection_flow_coefficient is an optional parameter that can be used to include the connection_flow variable from or to a node in a unit_constraint via the connection__from_node__unit_constraint and connection__to_node__unit_constraint relationships. Essentially, connection_flow_coefficient appears as a coefficient for the connection_flow variable from or to the node in the unit constraint.

connection_flow_cost

> Variable costs of a flow through a `connection`. E.g. EUR/MWh of energy throughput.

> **Related Object Classes:** connection

By defining the connection_flow_cost parameter for a specific `connection`, a cost term will be added to the objective function that values all `connection_flow` variables associated with that connection during the current optimization window.

connection_flow_delay

> Delays the `connection_flows` associated with the latter `node` in respect to the `connection_flows` associated with the first `node`.

> **Default value:** Dict{String, Any}("data" => "0h", "type" => "duration")

> **Related Relationship Classes:** connection__node__node

The connection_flow_delay parameter denotes the amount of time that it takes for the flow to go through a connection. In other words, the flow that enters the connection is only seen at the other side after connection_flow_delay units of time.

`connection_investment_cost`

> The per unit investment cost for the connection over the `connection_investment_lifetime`

> **Related Object Classes:** connection

By defining the connection_investment_cost parameter for a specific `connection`, a cost term will be added to the objective function whenever a connection investment is made during the current optimization window.

`connection_investment_lifetime`

> Determines the minimum investment lifetime of a connection. Once invested, it remains in service for this long

> **Related Object Classes:** connection

connection_investment_lifetime is the minimum amount of time that a connection has to stay in operation once it's invested-in. Only after that time, the connection can be decomissioned. Note that `connection_investment_lifetime` is a dynamic parameter that will impact the amount of solution history that must remain available to the optimisation in each step - this may impact performance.

`connection_investment_variable_type`

> Determines whether the investment variable is integer `variable_type_integer` or continuous `variable_type_continuous`

> **Default value:** variable_type_integer

> **Uses Parameter Value Lists:** variable_type_list

> **Related Object Classes:** connection

The connection_investment_variable_type parameter represents the *type* of the connections_invested_available decision variable.

The default value, `variable_type_integer`, means that only integer factors of the connection_capacity can be invested in. The value `variable_type_continuous` means that any fractional factor can also be invested in. The value `variable_type_binary` means that only a factor of 1 or zero are possible.

connection_linepack_constant

> The linepack constant is a property of gas pipelines and relates the linepack to the pressure of the adjacent nodes.

> **Related Relationship Classes:** connection__node__node

The linepack constant is a physical property of a connection representing a pipeline and holds information on how the linepack flexibility relates to pressures of the adjacent nodes. If, and only if, this parameter is defined, the linepack flexibility of a pipeline can be modelled. The existence of the parameter triggers the generation of the constraint on line pack storage. The connection_linepack_constant should always be defined on the tuple (connection pipeline, linepack storage node, node group (containing both pressure nodes, i.e. start and end of the pipeline)). See also.

connection_monitored

> A boolean flag for defining a contingency `connection`.

Default value: false

Uses Parameter Value Lists: boolean_value_list

Related Object Classes: connection

When using ptdf-based load flow by setting commodity_physics to either commodity_physics_ptdf or commodity_physics_ptdf, a constraint is created for each connection for which `connection_monitored` = `true`. Thus, to monitor the ptdf-based flow on a particular connection `connection_monitored` must be set to `true`.

See also powerflow

connection_reactance

The per unit reactance of a `connection`.

Related Object Classes: connection

The per unit reactance of a transmission line. Used in ptdf based dc load flow where the relative reactances of lines determine the ptdfs of the network and in lossless dc powerflow where the flow on a line is given by `flow = 1/x(theta_to-theta_from)` where x is the reatance of the line, theta*to is the voltage angle of the remote node and theta*from is the voltage angle of the sending node.

connection_reactance_base

If the reactance is given for a p.u. (e.g. p.u. = 100MW), the `connection_reactance_base` can be set to perform this conversion (e.g. *100).

Default value: 1

Related Object Classes: connection

As the connection_reactance is often given on a per unit basis, often different than the units used elsewhere, the connection_reactance_base parameter serves as a conversion factor, scaling the connection_reactance with its p.u..

connection_resistance

> The per unit resistance of a `connection`.

> **Related Object Classes:** connection

The per unit resistance of a transmission line. **Currently unimplemented!**

connection_type

> A selector between a normal and a lossless bidirectional `connection`.

> **Default value:** connection_type_normal

> **Uses Parameter Value Lists:** connection_type_list

> **Related Object Classes:** connection

Used to control specific pre-processing actions on connections. Currently, the primary purpose of `connection_type` is to simplify the data that is required to define a simple bi-directional, lossless line. If `connection_type` = `:connection_type_lossless_bidirectional`, it is only necessary to specify the following minimum data:

- relationship: connection__from_node
- relationship: connection__to_node
- parameter: connection_capacity (defined on connection__from_node and/or connection__to_node)

If `connection_type` = `:connection_type_lossless_bidirectional` the following pre-processing actions are taken:

- reciprocal connection__from_node and connection__to_node relationships are created if they don't exist
- a new connection__node__node relationship is created if none exists already
- fix_ratio_out_in_connection_flow parameter is created with the value of 1 if no existing parameter found (therefore this value can be overridden)
- The first connection_capacity parameter found is copied to connection__from_nodes and connection__to_nodes without a defined connection_capacity.

constraint_sense

> A selector for the sense of the `unit_constraint`.

> **Default value:** ==

> **Uses Parameter Value Lists:** constraint_sense_list

> **Related Object Classes:** unit_constraint

The constraint_sense parameter determines the *sense* of a custom user constraint.

See User constraints for details.

curtailment_cost

> Costs for curtailing generation. Essentially, accrues costs whenever `unit_flow` not operating at its maximum available capacity. E.g. EUR/MWh

> **Related Object Classes:** unit

By defining the curtailment_cost parameter for a specific `unit`, a cost term will be added to the objective function whenever this unit's available capacity exceeds its activity (i.e., the `unit_flow` variable) over the course of the operational dispatch during the current optimization window.

cyclic_condition

> If the cyclic condition is set to true for a storage node, the `node_state` at the end of the optimization window has to be larger than or equal to the initial storage state.

> **Default value:** false

> **Uses Parameter Value Lists:** boolean_value_list

> **Related Relationship Classes:** node__temporal_block

The cyclic_condition parameter is used to enforce that the storage level at the end of the optimization window is higher or equal to the storage level at the beginning optimization. If the cyclic_condition parameter is set to true for a node__temporal_block relationship, and the has_state parameter of the corrresponding node is set to true, the constraint_cyclic_node_state will be triggered.

`demand`

> Demand for the `commodity` of a `node`. Energy gains can be represented using negative `demand`.

> **Default value:** 0.0

> **Related Object Classes:** node

The demand parameter represents a "demand" or a "load" of a commodity on a node. It appears in the node injection constraint, with positive values interpreted as "demand" or "load" for the modelled system, while negative values provide the system with "influx" or "gain". When the node is part of a group, the fractional_demand parameter can be used to split demand into fractions, when desired. See also: Introduction to groups of objects

The demand parameter can also be included in custom unit_constraints using the demand_coefficient parameter for the node__unit_constraint relationship.

`demand_coefficient`

> coefficient of the specified node's demand in the specified unit constraint

> **Default value:** 0.0

> **Related Relationship Classes:** node__unit_constraint

The demand_coefficient is an optional parameter that can be used to include the demand of the a node in a unit_constraint via the node__unit_constraint relationship. Essentially, demand_coefficient appears as a coefficient for the demand parameter of the connected node in the unit constraint.

`diff_coeff`

> Commodity diffusion coefficient between two `nodes`. Effectively, denotes the *diffusion power per unit of state* from the first `node` to the second.

> **Default value:** 0.0

> **Related Relationship Classes:** node__node

The diff_coeff parameter represents diffusion of a commodity between the two nodes in the node__node relationship. It appears as a coefficient on the `node_state` variable in the node injection constraint, essentially representing *diffusion power per unit of state*. Note that the diff_coeff is interpreted as *one-directional*, meaning that if one defines

```
diff_coeff(node1=n1, node2=n2),
```

there will only be diffusion from `n1` to `n2`, but not vice versa. *Symmetric diffusion* is likely used in most cases, requiring defining the diff_coeff both ways

```
diff_coeff(node1=n1, node2=n2) == diff_coeff(node1=n2, node2=n1).
```

`downward_reserve`

> Identifier for `nodes` providing downward reserves

> **Default value:** false

> **Related Object Classes:** node

If a node has a `true` is_reserve_node parameter, it will be treated as a reserve node in the model. To define whether the node corresponds to an upward or downward reserve commodity, the upward_reserve or the downward_reserve parameter needs to be set to true, respectively.

`duration_unit`

> Defines the base temporal unit of the `model`. Currently supported values are either an `hour` or a `minute`.

> **Default value:** minute

> **Uses Parameter Value Lists:** duration_unit_list

> **Related Object Classes:** model

The duration_unit parameter specifies the base unit of time in a model. Two values are currently supported, `hour` and the default `minute`. E.g. if the duration_unit is set to `hour`, a `Duration` of one `minute` gets converted into `1/60 hours` for the calculations.

`fix_binary_gas_connection_flow`

> Fix the value of the `connection_flow_binary` variable, and hence pre-determine the direction of flow in the connection.

> **Related Relationship Classes:** connection__from_node

The binary flow of a gas pipelines for pressure driven gas transfer is enables through the binary variable binary_gas_connection_flow and the big_m constant. To fix this binary variable, i.e. pre-define the direction of gas through the pipelines, the fix_binary_gas_connection_flow parameter can be used.

`fix_connection_flow`

> Fix the value of the `connection_flow` variable.

> **Related Relationship Classes:** connection__from_node and connection__to_node

The fix_connection_flow parameter fixes the value of the connection_flow variable.

`fix_connection_intact_flow`

> Fix the value of the `connection_intact_flow` variable.

> **Related Relationship Classes:** connection__from_node and connection__to_node

The fix_connection_intact_flow parameter can be used to fix the values of the connection_intact_flow variable to preset values. If set to a `Scalar` type value, the connection_intact_flow variable is fixed to that value for all time steps and stochastic_scenarios. Values for individual time steps can be fixed using `TimeSeries` type values.

`fix_connections_invested`

> Setting a value fixes the connections_invested variable accordingly

> **Related Object Classes:** connection

The fix_connections_invested parameter can be used to fix the values of the connections_invested variable to preset values. If set to a `Scalar` type value, the connections_invested variable is fixed to that value for all time steps and stochastic_scenarios. Values for individual time steps can be fixed using `TimeSeries` type values.

See Investment Optimization for more information about the investment framework in *SpineOpt.jl*.

fix_connections_invested_available

> Setting a value fixes the connections *invested* available variable accordingly

> **Related Object Classes:** connection

The fix_connections_invested_available parameter represents a *forced* connection investment.

In other words, it is the fix value of the connections_invested_available variable.

fix_node_pressure

> Fixes the corresponding `node_pressure` variable to the provided value

> **Related Object Classes:** node

In a pressure driven gas model, gas network nodes are associated with the node_pressure variable. In order to fix the pressure at a certain node or to give intial conditions the fix_node_pressure parameter can be used.

fix_node_state

> Fixes the corresponding `node_state` variable to the provided value. Can be used for e.g. fixing boundary conditions.

> **Related Object Classes:** node

The fix_node_state parameter simply fixes the value of the `node_state` variable to the provided value, if one is found. Common uses for the parameter include e.g. providing initial values for `node_state` variables, by fixing the value on the first modelled time step *(or the value before the first modelled time step)* using a `TimeSeries` type parameter value with an appropriate timestamp. Due to the way *SpineOpt* handles `TimeSeries` data, the `node_state` variables are only fixed for time steps with defined fix_node_state parameter values.

fix_node_voltage_angle

> Fixes the corresponding `node_voltage_angle` variable to the provided value

> **Related Object Classes:** node

For a lossless nodal DC power flow network, each node is associated with a node_voltage_angle variable. In order to fix the voltage angle at a certain node or to give initial conditions the fix_node_voltage_angle parameter can be used.

fix_nonspin_ramp_down_unit_flow

> Fix the `nonspin_ramp_down_unit_flow` variable.

> **Related Relationship Classes:** unit__to_node

The `fix_nonspin_ramp_down_unit_flow` parameter simply fixes the value of the nonspin_ramp_down_unit_flow variable to the provided value. As such, it determines directly how much non-spinning downward reserve commodity flows the relevant unit is providing to the node to which it is linked by the unit__to_node relationship.

When a single value is selected, this value is kept constant throughout the model. It is also possible to provide a timeseries of values, which can be used for example to impose initial conditions by providing a value only for the first timestep included in the model.

fix_nonspin_ramp_up_unit_flow

> Fix the `nonspin_ramp_up_unit_flow` variable.

> **Related Relationship Classes:** unit__from_node and unit__to_node

The `fix_nonspin_ramp_up_unit_flow` parameter simply fixes the value of the nonspin_ramp_up_unit_flow variable to the provided value. As such, it determines directly how much non-spinning upward reserve commodity flows the relevant unit is providing to the node to which it is linked by the unit__to_node relationship.

When a single value is selected, this value is kept constant throughout the model. It is also possible to provide a timeseries of values, which can be used for example to impose initial conditions by providing a value only for the first timestep included in the model.

`fix_nonspin_units_shut_down`

> Fix the `nonspin_units_shut_down` variable.

> **Related Relationship Classes:** unit__to_node

The `fix_nonspin_units_shut_down` parameter simply fixes the value of the nonspin_units_shut_down variable to the provided value. As such, it determines directly how many member units are involved in providing downward reserve commodity flows to the node to which it is linked by the unit__to_node relationship.

When a single value is selected, this value is kept constant throughout the model. It is also possible to provide a timeseries of values, which can be used for example to impose initial conditions by providing a value only for the first timestep included in the model.

`fix_nonspin_units_started_up`

> Fix the `nonspin_units_started_up` variable.

> **Related Relationship Classes:** unit__from_node and unit__to_node

The `fix_nonspin_units_started_up` parameter simply fixes the value of the nonspin_units_started_up variable to the provided value. As such, it determines directly how many member units are involved in providing upward reserve commodity flows to the node to which it is linked by the unit__to_node relationship.

When a single value is selected, this value is kept constant throughout the model. It is also possible to provide a timeseries of values, which can be used for example to impose initial conditions by providing a value only for the first timestep included in the model.

`fix_ramp_down_unit_flow`

> Fix the `ramp_down_unit_flow` variable.

> Related Relationship Classes: unit__to_node

The `fix_ramp_down_unit_flow` parameter simply fixes the value of the ramp_down_unit_flow variable to the provided value. It is possible to provide an incomplete timeseries of values, which can be used for example to impose initial conditions by providing a value only for the first timestep included in the model.

`fix_ramp_up_unit_flow`

> Fix the `ramp_up_unit_flow` variable.

> Related Relationship Classes: unit__from_node and unit__to_node

The `fix_ramp_up_unit_flow` parameter simply fixes the value of the ramp_up_unit_flow variable to the provided value. It is possible to provide an incomplete timeseries of values, which can be used for example to impose initial conditions by providing a value only for the first timestep included in the model.

`fix_ratio_in_in_unit_flow`

> Fix the ratio between two `unit_flows` coming into the `unit` from the two `nodes`.

> Related Relationship Classes: unit__node__node

The definition of the fix_ratio_in_in_unit_flow parameter triggers the generation of the constraint_fix_ratio_in_in_unit_flow and fixes the ratio between incoming flows of a unit. The parameter is defined on the relationship class unit__node__node, where both nodes (or group of nodes) in this relationship represent `from_node`s, i.e. the incoming flows to the unit. The ratio parameter is interpreted such that it constrains the ratio of `in1` over `in2`, where `in1` is the unit_flow variable from the first node in the unit__node__node relationship in a left-to-right order. This parameter can be useful, for instance if a unit requires a specific commodity mix as a fuel supply.

To enforce e.g. for a unit `u` a fixed share of `0.8` of its incoming flow from the node `supply_fuel_1` compared to its incoming flow from the node group `supply_fuel_2` (consisting of the two nodes

`supply_fuel_2_component_a` and `supply_fuel_2_component_b`) the fix_ratio_in_in_unit_flow parameter would be set to `0.8` for the relationship `u__supply_fuel_1__supply_fuel_2`.

fix_ratio_in_out_unit_flow

> Fix the ratio between an incoming `unit_flow` from the first `node` and an outgoing `unit_flow` to the second `node`.

> **Related Relationship Classes:** unit__node__node

The definition of the fix_ratio_in_out_unit_flow parameter triggers the generation of the constraint_fix_ratio_in_out_unit_flow and fixes the ratio between incoming and outgoing flows of a unit. The parameter is defined on the relationship class unit__node__node, where the first node (or group of nodes) in this relationship represents the `from_node`,i i.e. the incoming flows to the unit, and the second node (or group of nodes), represents the `to_node` i.e. the outgoing flow from the unit. The ratio parameter is interpreted such that it constrains the ratio of `in` over `out`, where `in` is the unit_flow variable from the first node in the unit__node__node relationship in a left-to-right order.

To enforce e.g. a fixed ratio of `1.4` for a unit `u` between its incoming gas flow from the node `ng` and its outgoing flows to the node group `el_heat` (consisting of the two nodes `el` and `heat`), the fix_ratio_in_out_unit_flow parameter would be set to `1.4` for the relationship `u__ng__el_heat`.

fix_ratio_out_in_connection_flow

> Fix the ratio between the `connection_flow` from the first `node` and the `connection_flow` to the second `node`.

> **Related Relationship Classes:** connection__node__node

The definition of the fix_ratio_out_in_connection_flow parameter triggers the generation of the constraint_fix_ratio_out_in_connection_flow and fixes the ratio between outgoing and incoming flows of a connection. The parameter is defined on the relationship class connection__node__node, where the first node (or group of nodes) in this relationship represents the `to_node`, i.e. the outgoing flow from the connection, and the second node (or group of nodes), represents the `from_node`, i.e. the incoming flows to the connection. In most cases the fix_ratio_out_in_connection_flow parameter is set to equal or lower than `1`, linking the flows entering to the flows leaving the connection. The ratio parameter is interpreted such that it constrains the ratio of `out` over `in`, where `out` is the connection_flow variable

from the first node in the connection__node__node relationship in a left-to-right order. The parameter can be used to e.g. account for losses over a connection in a certain direction.

To enforce e.g. a fixed ratio of `0.8` for a connection `conn` between its outgoing electricity flow to node `el1` and its incoming flows from the node node `el2`, the fix_ratio_out_in_connection_flow parameter would be set to `0.8` for the relationship `u__el1__el2`.

fix_ratio_out_in_unit_flow

> Fix the ratio between an outgoing `unit_flow` to the first `node` and an incoming `unit_flow` from the second `node`.

> **Related Relationship Classes:** unit__node__node

The definition of the fix_ratio_out_in_unit_flow parameter triggers the generation of the constraint_fix_ratio_out_in_unit_flow and fixes the ratio between out and incoming flows of a unit. The parameter is defined on the relationship class unit__node__node, where the first node (or group of nodes) in this relationship represents the `to_node`, i.e. the outgoing flow from the unit, and the second node (or group of nodes), represents the `from_node`, i.e. the incoming flows to the unit. The ratio parameter is interpreted such that it constrains the ratio of `out` over `in`, where `out` is the unit_flow variable from the first node in the unit__node__node relationship in a left-to-right order.

To enforce e.g. a fixed ratio of `0.8` for a unit `u` between its outgoing flows to the node group `el_heat` (consisting of the two nodes `el` and `heat`) and its incoming gas flow from `ng`the fix_ratio_out_in_unit_flow parameter would be set to `0.8` for the relationship `u__el_heat__ng`.

fix_ratio_out_out_unit_flow

> Fix the ratio between two `unit_flows` going from the `unit` into the two `nodes`.

> **Related Relationship Classes:** unit__node__node

The definition of the fix_ratio_out_out_unit_flow parameter triggers the generation of the constraint_fix_ratio_out_out_unit_flow and fixes the ratio between outgoing flows of a unit. The parameter is defined on the relationship class unit__node__node, where the nodes (or group of nodes) in this relationship represent the `to_node`'s', i.e. outgoing flow from the unit. The ratio parameter is interpreted such that it constrains the ratio of `out1` over `out2`, where `out1` is the unit_flow variable from the first node in the unit__node__node relationship in a left-to-right reading order.

To enforce a fixed ratio between two products of a unit `u`, e.g. fixing the share of produced electricity flowing to node `el` to `0.4` of the production of heat flowing to node `heat`, the fix_ratio_out_out_unit_flow parameter would be set to `0.4` for the relationship `u__el__heat`.

fix_shut_down_unit_flow

> Fix the `shut_down_unit_flow` variable.

> **Related Relationship Classes:** unit__to_node

The fix_shut_down_unit_flow parameter fixes the value of the shut_down_unit_flow to the provided value, if the parameter is defined.

Common uses for the parameter include e.g. providing initial values for the shut_down_unit_flow, by fixing the value on the first modelled time step *(or the value before the first modelled time step)* using a `TimeSeries` type parameter value with an appropriate timestamp. Due to the way *SpineOpt* handles `TimeSeries` data, the shut_down_unit_flow variable is only fixed for time steps with defined fix_shut_down_unit_flow parameter values.

Other uses can include e.g. a constant or time-varying **exogenous** commodity flow from or to a unit.

Note that the mentioned shut_down_unit_flow variable is only included if the parameter max_startup_ramp exist for the correspond unit__to_node or unit__from_node relationship. The usage of ramps is described in Ramping and Reserves.

fix_start_up_unit_flow

> Fix the `start_up_unit_flow` variable.

> **Related Relationship Classes:** unit__from_node and unit__to_node

The fix_start_up_unit_flow parameter fixes the value of the start_up_unit_flow to the provided value, if the parameter is defined.

Common uses for the parameter include e.g. providing initial values for the start_up_unit_flow, by fixing the value on the first modelled time step *(or the value before the first modelled time step)* using a `TimeSeries` type parameter value with an appropriate timestamp. Due to the way *SpineOpt* handles `TimeSeries` data, the start_up_unit_flow variable is only fixed for time steps with defined fix_start_up_unit_flow parameter values.

Other uses can include e.g. a constant or time-varying **exogenous** commodity flow from or to a unit.

Note that the mentioned start_up_unit_flow variable is only included if the parameter max_startup_ramp exist for the correspond unit__to_node or unit__from_node relationship. The usage of ramps is described in Ramping and Reserves.

`fix_storages_invested`

> Used to fix the value of the storages_invested variable

> **Related Object Classes:** node

Used primarily to fix the value of the `storages_invested` variable which represents the point-in-time storage investment decision variable at a node and how many candidate storages are invested-in in a particular timeslice at the corresponding node.

See also Investment Optimization, candidate_storages and storage_investment_variable_type

`fix_storages_invested_available`

> Used to fix the value of the storages*invested*available variable

> **Related Object Classes:** node

Used primarily to fix the value of the `storages_invested_available` variable which represents the storages investment decision variable and how many candidate storages are available at the corresponding node, time step and stochastic scenario. Used also in the decomposition framework to communicate the value of the master problem solution variables to the operational sub-problem.

See also candidate_storages and Investment Optimization

`fix_unit_flow`

> Fix the `unit_flow` variable.

> **Related Relationship Classes:** unit__from_node and unit__to_node

The fix_unit_flow parameter fixes the value of the unit_flow variable to the provided value, if the parameter is defined.

Common uses for the parameter include e.g. providing initial values for the unit_flow variable, by fixing the value on the first modelled time step *(or the value before the first modelled time step)* using a `TimeSeries` type parameter value with an appropriate timestamp. Due to the way *SpineOpt* handles `TimeSeries` data, the unit_flow variable is only fixed for time steps with defined fix_unit_flow parameter values.

Other uses can include e.g. a constant or time-varying **exogenous** commodity flow from or to a unit.

`fix_unit_flow_op`

> Fix the `unit_flow_op` variable.

> **Related Relationship Classes:** unit__from_node and unit__to_node

If operating_points is defined on a certain `unit__to_node` or `unit__from_node` flow, the corresponding `unit_flow` flow variable is decomposed into a number of sub-variables, `unit_flow_op` one for each operating point, with an additional index, `i` to reference the specific operating point. `fix_unit_flow_op` can thus be used to fix the value of one or more of the variables as desired.

`fix_units_invested`

> Fix the value of the `units_invested` variable.

> **Related Object Classes:** unit

Used primarily to fix the value of the `units_invested` variable which represents the point-in-time unit investment decision variable and how many candidate units are invested-in in a particular timeslice.

See also Investment Optimization, candidate_units and unit_investment_variable_type

`fix_units_invested_available`

> Fix the value of the `units_invested_available` variable

> **Related Object Classes:** unit

Used primarily to fix the value of the `units_invested_available` variable which represents the unit investment decision variable and how many candidate units are invested-in and available at the corresponding node, time step and stochastic scenario. Used also in the decomposition framework to communicate the value of the master problem solution variables to the operational sub-problem.

See also Investment Optimization, candidate_units and unit_investment_variable_type

`fix_units_on`

> Fix the value of the `units_on` variable.

> **Related Object Classes:** unit

The `fix_units_on` parameter simply fixes the value of the units_on variable to the provided value. As such, it determines directly how many members of the specific unit will be online throughout the model when a single value is selected. It is also possible to provide a timeseries of values, which can be used for example to impose initial conditions by providing a value only for the first timestep included in the model.

`fix_units_on_coefficient_in_in`

> Optional coefficient for the `units_on` variable impacting the `fix_ratio_in_in_unit_flow` constraint.

> **Default value:** 0.0

> **Related Relationship Classes:** unit__node__node

The fix_units_on_coefficient_in_in parameter is an optional coefficient in the unit input-input ratio constraint controlled by the fix_ratio_in_in_unit_flow parameter. Essentially, it acts as a coefficient for the units_on variable in the constraint, allowing for fixing the conversion ratio depending on the amount of online capacity.

Note that there are different parameters depending on the directions of the unit_flow variables being constrained: fix_units_on_coefficient_in_out, fix_units_on_coefficient_out_in, and fix_units_on_coefficient_out_out, all of which apply to their respective constraints. Similarly, there are different parameters for setting minimum or maximum conversion rates, e.g. min_units_on_coefficient_in_in and max_units_on_coefficient_in_in.

`fix_units_on_coefficient_in_out`

> Optional coefficient for the `units_on` variable impacting the `fix_ratio_in_out_unit_flow` constraint.

> **Default value:** 0.0

> **Related Relationship Classes:** unit__node__node

The fix_units_on_coefficient_in_out parameter is an optional coefficient in the unit input-output ratio constraint controlled by the fix_ratio_in_out_unit_flow parameter. Essentially, it acts as a coefficient for the units_on variable in the constraint, allowing for fixing the conversion ratio depending on the amount of online capacity.

Note that there are different parameters depending on the directions of the unit_flow variables being constrained: fix_units_on_coefficient_in_in, fix_units_on_coefficient_out_in, and fix_units_on_coefficient_out_out, all of which apply to their respective constraints. Similarly, there are different parameters for setting minimum or maximum conversion rates, e.g. min_units_on_coefficient_in_out and max_units_on_coefficient_in_out.

`fix_units_on_coefficient_out_in`

> Optional coefficient for the `units_on` variable impacting the `fix_ratio_out_in_unit_flow` constraint.

> **Default value:** 0.0

> **Related Relationship Classes:** unit__node__node

The fix_units_on_coefficient_out_in parameter is an optional coefficient in the unit output-input ratio constraint controlled by the fix_ratio_out_in_unit_flow parameter. Essentially, it acts as a coefficient for the units_on variable in the constraint, allowing for fixing the conversion ratio depending on the amount of online capacity.

Note that there are different parameters depending on the directions of the unit_flow variables being constrained: fix_units_on_coefficient_in_in, fix_units_on_coefficient_in_out, and fix_units_on_coefficient_out_out, all of which apply to their respective constraints. Similarly, there are different parameters for setting minimum or maximum conversion rates, e.g. min_units_on_coefficient_out_in and max_units_on_coefficient_out_in.

`fix_units_on_coefficient_out_out`

> Optional coefficient for the `units_on` variable impacting the `fix_ratio_out_out_unit_flow` constraint.

> **Default value:** 0.0

> **Related Relationship Classes:** unit__node__node

The fix_units_on_coefficient_out_out parameter is an optional coefficient in the unit output-output ratio constraint controlled by the fix_ratio_out_out_unit_flow parameter. Essentially, it acts as a coefficient for the units_on variable in the constraint, allowing for fixing the conversion ratio depending on the amount of online capacity.

Note that there are different parameters depending on the directions of the unit_flow variables being constrained: fix_units_on_coefficient_in_in, fix_units_on_coefficient_in_out, and fix_units_on_coefficient_out_in, all of which apply to their respective constraints. Similarly, there are different parameters for setting minimum or maximum conversion rates, e.g. min_units_on_coefficient_out_out and max_units_on_coefficient_out_out.

`fixed_pressure_constant_0`

> Fixed pressure points for pipelines for the outer approximation of the Weymouth approximation. The direction of flow is the first node in the relationship to the second node in the relationship.

> Related Relationship Classes: connection__node__node

For the MILP representation of pressure driven gas transfer, we use an outer approximation approach as described by Schwele et al.. The Weymouth equation is approximated around fixed pressure points, as described by the constraint on fixed node pressure points, constraining the average flow in each direction dependent on the adjacent node pressures. The second fixed pressure constant, which will be multiplied with the pressure of the destination node, is represented by an Array value of the fixed_pressure_constant_0. The first pressure constant corresponds to the related parameter fixed_pressure_constant_1. Note that the fixed_pressure_constant_0 parameter should be defined on a connection__node__node relationship, for which the first node corresponds to the origin node, while the second node corresponds to the destination node. For a typical gas pipeline, the will be a fixed_pressure_constant_1 for both directions of flow.

`fixed_pressure_constant_1`

> Fixed pressure points for pipelines for the outer approximation of the Weymouth approximation. The direction of flow is the first node in the relationship to the second node in the relationship.

> Related Relationship Classes: connection__node__node

For the MILP representation of pressure driven gas transfer, we use an outer approximation approach as described by Schwele et al.. The Weymouth equation is approximated around fixed pressure points, as described by the constraint on fixed node pressure points, constraining the average flow in each direction dependent on the adjacent node pressures. The first fixed pressure constant, which will be multiplied with the pressure of the origin node, is represented by an Array value of the fixed_pressure_constant_1. The second pressure constant corresponds to the related parameter fixed_pressure_constant_0. Note that the fixed_pressure_constant_1 parameter should be defined on a connection__node__node relationship, for which the first node corresponds to the origin node, while the second node corresponds to the destination node. For a typical gas pipeline, the will be a fixed_pressure_constant_1 for both directions of flow.

`fom_cost`

> Fixed operation and maintenance costs of a `unit`. Essentially, a cost coefficient on the `number_of_units` and `unit_capacity` parameters. E.g. EUR/MWh

> **Related Object Classes:** unit

By defining the fom_cost parameter for a specific `unit`, a cost term will be added to the objective function to account for the fixed operation and maintenance costs associated with that unit during the current optimization window.

`frac_state_loss`

> Self-discharge coefficient for `node_state` variables. Effectively, represents the *loss power per unit of state*.

> **Default value:** 0.0

> **Related Object Classes:** node

The frac_state_loss parameter allows setting self-discharge losses for nodes with the `node_state` variables enabled using the has_state variable. Effectively, the frac_state_loss parameter acts as a coefficient on the `node_state` variable in the node injection constraint, imposing losses for the node. In simple cases, storage losses are typically fractional, e.g. a frac_state_loss parameter value of 0.01 would represent 1% of `node_state` lost per unit of time. However, a more general definition of what the frac_state_loss parameter represents in *SpineOpt* would be *loss power per unit of* `node_state`.

`fractional_demand`

> The fraction of a `node` group's `demand` applied for the `node` in question.

> **Default value:** 0.0

> **Related Object Classes:** node

Whenever a node is a member of a group, the fractional_demand parameter represents its share of the group's demand.

`fuel_cost`

> Variable fuel costs than can be attributed to a `unit_flow`. E.g. EUR/MWh

> **Related Relationship Classes:** unit__from_node and unit__to_node

By defining the fuel_cost parameter for a specific `unit`, `node`, and `direction`, a cost term will be added to the objective function to account for costs associated with the unit's fuel usage over the course of its operational dispatch during the current optimization window.

`graph_view_position`

> An optional setting for tweaking the position of the different elements when drawing them via Spine Toolbox Graph View.

> **Related Relationship Classes:** connection__from_node, connection__to_node, unit__from_node__unit_constraint, unit__from_node, unit__to_node__unit_constraint and unit__to_node

The graph_view_position parameter can be used to fix the positions of various objects and relationships when plotted using the *Spine Toolbox Graph View*. If not defined, *Spine Toolbox* simply plots the element in question wherever it sees fit in the graph.

`has_binary_gas_flow`

> This parameter needs to be set to `true` in order to represent bidirectional pressure drive gas transfer.

> **Default value:** false

> **Uses Parameter Value Lists:** boolean_value_list

> **Related Object Classes:** connection

This parameter is necessary for the use of pressure driven gas transfer, for which the direction of flow is not known a priori. The parameter has_binary_gas_flow is a booelean method parameter, which - when set to true - triggers the generation of the binary variables binary_gas_connection_flow, which (together with the big_m parameter) forces the average flow through a pipeline to be unidirectional.

`has_pressure`

> A boolean flag for whether a `node` has a `node_pressure` variable.

> **Default value:** false

> **Uses Parameter Value Lists:** boolean_value_list

> **Related Object Classes:** node

If a node is to represent a node in a pressure driven gas network, the boolean parameter has_pressure should be set true, in order to trigger the generation of the node_pressure variable. The pressure at a certain node can also be constrainted through the parameters max_node_pressure and min_node_pressure. More details on the use of pressure driven gas transfer are described here

`has_state`

> A boolean flag for whether a `node` has a `node_state` variable.

> **Default value:** false

> **Uses Parameter Value Lists:** boolean_value_list

> **Related Object Classes:** node

The has_state parameter is simply a `Bool` flag for whether a node has a node_state variable. By default, it is set to `false`, so the nodes enforce instantaneous commodity balance according to the nodal balance and node injection constraints. If set to `true`, the node will have a node_state variable generated for it, allowing for commodity storage at the node. Note that you'll also have to specify a value for the state_coeff parameter, as otherwise the node_state variable has zero commodity capacity.

`has_voltage_angle`

> A boolean flag for whether a `node` has a `node_voltage_angle` variable.

> **Default value:** false

> **Uses Parameter Value Lists:** boolean_value_list

> **Related Object Classes:** node

For the use of node-based lossless DC powerflow, each node will be associated with a node_voltage_angle variable. To enable the generation of the variable in the optimization model, the boolean parameter has_voltage_angle should be set true. The voltage angle at a certain node can also be constrained through the parameters max_voltage_angle and min_voltage_angle. More details on the use of lossless nodal DC power flows are described here

`is_active`

> If false, the object is excluded from the model if the tool filter object activity control is specified

> **Default value:** true

> **Uses Parameter Value Lists:** boolean_value_list

> **Related Object Classes:** commodity, connection, model, node, output, report, stochastic_scenario, stochastic_structure, temporal_block, unit_constraint and unit

`is_acive` is a universal, utility parameter that is defined for every object class. When used in conjunction with the `activity_control` feature, the `is_active` parameter allows one to control whether or not a specific object is active within a model or not.

`is_reserve_node`

> A boolean flag for whether a `node` is acting as a `reserve_node`

> **Default value:** false

> **Uses Parameter Value Lists:** boolean_value_list

> **Related Object Classes:** node

By setting the parameter is_reserve_node to `true`, a node is treated as a reserve node in the model. Units that are linked through a unit__to_node relationship will be able to provide balancing services to the reserve node, but within their technical feasibility. The mathematical formulation holds a chapter on Ramping and reserve constraints and the general concept of setting up a model with reserves is described in Ramping and Reserves.

`max_cum_in_unit_flow_bound`

> Set a maximum cumulative upper bound for a `unit_flow`

> **Related Relationship Classes:** unit__commodity

To impose a limit on the cumulative in flows to a unit for the entire modelling horizon, e.g. to enforce limits on emissions, the max_cum_in_unit_flow_bound parameter can be used. Defining this parameter triggers the generation of the constraint_max_cum_in_unit_flow_bound.

Assuming for instance that the total intake of a unit `u_A` should not exceed `10MWh` for the entire modelling horizon, then the max_cum_in_unit_flow_bound would need to take the value `10`. (Assuming here that the unit_flow variable is in `MW`, and the model duration_unit is hours)

`max_gap`

> Specifies the maximum optimality gap for the model. Currently only used for the master problem within a decomposed structure

> **Default value:** 0.05

> **Related Object Classes:** model

This determines the optimality convergence criterion and is the benders gap tolerance for the master problem in a decomposed investments model. The benders gap is the relative difference between the current objective function upper bound(z*upper) and lower bound (z*lower) and is defined as 2*(z*upper-z*lower)/(z*upper + z*lower). When this value is lower than `max_gap` the benders algorithm will terminate having achieved satisfactory optimality.

`max_iterations`

> Specifies the maximum number of iterations for the model. Currently only used for the master problem within a decomposed structure

> **Default value:** 10.0

> **Related Object Classes:** model

When the model in question is of type `:spineopt_master`, this determines the maximum number of Benders iterations.

`max_node_pressure`

> Maximum allowed gas pressure at `node`.

> **Related Object Classes:** node

If a node has a node_pressure variable (see also the parameter has_pressure and this chapter), an upper bound on the pressure can be introduced through the max_node_pressure parameter, which triggers the generation of the maxmimum node pressure constraint.

`max_ratio_in_in_unit_flow`

> Maximum ratio between two `unit_flows` coming into the `unit` from the two `nodes`.

> **Related Relationship Classes:** unit__node__node

The definition of the max_ratio_in_in_unit_flow parameter triggers the generation of the constraint_max_ratio_in_in_unit_flow and enforces an upper bound on the ratio between incoming flows of a unit. The parameter is defined on the relationship class unit__node__node, where both nodes (or group of nodes) in this relationship represent `from_nodes`, i.e. the incoming flows to the unit. The ratio parameter is interpreted such that it constrains the ratio of `in1` over `in2`, where `in1` is the unit_flow variable from the first node in the unit__node__node relationship in a left-to-right reading order. This parameter can be useful, for instance if a unit requires a specific commodity mix as a fuel supply.

To enforce e.g. for a unit `u` a maximum share of `0.8` of its incoming flow from the node `supply_fuel_1` compared to its incoming flow from the node group `supply_fuel_2` (consisting of the two nodes `supply_fuel_2_component_a` and `supply_fuel_2_component_b`) the max_ratio_in_in_unit_flow parameter would be set to `0.8` for the relationship `u__supply_fuel_1__supply_fuel_2`.

`max_ratio_in_out_unit_flow`

> Maximum ratio between an incoming `unit_flow` from the first `node` and an outgoing `unit_flow` to the second `node`.

> **Related Relationship Classes:** unit__node__node

The definition of the max_ratio_in_out_unit_flow parameter triggers the generation of the constraint_max_ratio_in_out_unit_flow and sets an upper bound on the ratio between incoming and outgoing flows of a unit. The parameter is defined on the relationship class unit__node__node, where the first node (or group of nodes) in this relationship represents the `from_node`, i.e. the incoming flows to the unit, and the second node (or group of nodes), represents the `to_node` i.e. the outgoing flow from the unit. The ratio parameter is interpreted such that it constrains the ratio of `in` over `out`, where `in` is the unit_flow variable from the first node in the unit__node__node relationship in a left-to-right reading order.

To enforce e.g. a maximum ratio of `1.4` for a unit `u` between its incoming gas flow from the node `ng` and its outgoing flow to the node group `el_heat` (consisting of the two nodes `el` and `heat`), the max_ratio_in_out_unit_flow parameter would be set to `1.4` for the relationship `u__ng__el_heat`.

max_ratio_out_in_connection_flow

> Maximum ratio between the `connection_flow` from the first `node` and the `connection_flow` to the second `node`.

> **Related Relationship Classes:** connection__node__node

The definition of the max_ratio_out_in_connection_flow parameter triggers the generation of the constraint_max_ratio_out_in_connection_flow and sets an upper bound on the ratio between outgoing and incoming flows of a connection. The parameter is defined on the relationship class connection__node__node, where the first node (or group of nodes) in this relationship represents the `to_node`, i.e. the outgoing flow from the connection, and the second node (or group of nodes), represents the `from_node`, i.e. the incoming flows to the connection. The ratio parameter is interpreted such that it constrains the ratio of `out` over `in`, where `out` is the connection_flow variable from the first node in the connection__node__node relationship in a left-to-right reading order.

To enforce e.g. a maximum ratio of `0.8` for a connection `conn` between its outgoing electricity flow to node `commodity1` and its incoming flows from the node node `commodity2`, the max_ratio_out_in_connection_flow parameter would be set to `0.8` for the relationship `conn__commodity1__commodity2`.

Note that the ratio can also be defined for connection__node__node relationships where one or both of the nodes correspond to node groups in order to impose a ratio on aggregated connection flows.

max_ratio_out_in_unit_flow

> Maximum ratio between an outgoing `unit_flow` to the first `node` and an incoming `unit_flow` from the second `node`.

> **Related Relationship Classes:** unit__node__node

The definition of the max_ratio_out_in_unit_flow parameter triggers the generation of the constraint_max_ratio_out_in_unit_flow and enforces an upper bound on the ratio between outgoing and incoming flows of a unit. The parameter is defined on the relationship class unit__node__node, where the first node (or group of nodes) in this relationship represents the `to_node`, i.e. the outgoing flow from the unit, and the second node (or group of nodes), represents the `from_node`, i.e. the incoming flows to the unit. The ratio parameter is interpreted such that it constrains the ratio of `out` over `in`, where `out` is the unit_flow variable from the first node in the unit__node__node relationship in a left-to-right reading order.

To enforce e.g. a maximum ratio of `0.8` for a unit `u` between its outgoing flows to the node group `el_heat` (consisting of the two nodes `el` and `heat`) and its incoming gas flow from `ng` the max_ratio_out_in_unit_flow parameter would be set to `0.8` for the relationship `u__el_heat__ng`.

max_ratio_out_out_unit_flow

> Maximum ratio between two `unit_flows` going from the `unit` into the two `nodes`.

> **Related Relationship Classes:** unit__node__node

The definition of the max_ratio_out_out_unit_flow parameter triggers the generation of the constraint_max_ratio_out_out_unit_flow and sets an upper bound on the ratio between outgoing flows of a unit. The parameter is defined on the relationship class unit__node__node, where the nodes (or group of nodes) in this relationship represent the `to_node`'s', i.e. outgoing flow from the unit. The ratio parameter is interpreted such that it constrains the ratio of `out1` over `out2`, where `out1` is the unit_flow variable from the first node in the unit__node__node relationship in a left-to-right reading order.

To enforce a maximum ratio between two products of a unit `u`, e.g. setting the maximum share of produced electricity flowing to node `el` to `0.4` of the production of heat flowing to node `heat`, the fix_ratio_out_out_unit_flow parameter would be set to `0.4` for the relationship `u__el__heat`.

max_res_shutdown_ramp

> Maximum non-spinning reserve ramp-down for online units providing reserves during shut-downs

> **Related Relationship Classes:** unit__from_node and unit__to_node

A unit can provide spinning and nonspinning reserves to a reserve node. These reserves can be either upward_reserve or downward_reserve. Nonspinning downward reserves are provided to a downward_reserve node by contracted units holding available to shutdown. To include the provision of nonspinning downward reserves, the parameter max_res_shutdown_ramp needs to be defined on the corresponding unit__to_node relationship. This will trigger the generation of the variables nonspin_units_shut_down and nonspin_ramp_down_unit_flow and the constraint on maximum downward nonspinning reserve provision. Note that max_res_shutdown_ramp is given as a fraction of the unit_capacity.

A detailed description of the usage of ramps and reserves is given in the chapter Ramping and Reserves. The chapter Ramping and reserve constraints in the Mathematical Formulation presents the equations related to ramps and reserves.

`max_res_startup_ramp`

> Maximum non-spinning reserve ramp-up for offline units scheduled for reserve provision

> **Related Relationship Classes:** unit__from_node and unit__to_node

A unit can provide spinning and nonspinning reserves to a reserve node. These reserves can be either upward_reserve or downward_reserve. Nonspinning upward reserves are provided to a upward_reserve node by contracted offline units holding available to startup. To include the provision of nonspinning upward reserves, the parameter max_res_startup_ramp needs to be defined on the corresponding unit__to_node relationship. This will trigger the generation of the variables nonspin_units_started_up and nonspin_ramp_up_unit_flow and the constraint on maximum upward nonspinning reserve provision. Note that max_res_startup_ramp is given as a fraction of the unit_capacity.

A detailed description of the usage of ramps and reserves is given in the chapter Ramping and Reserves. The chapter Ramping and reserve constraints in the Mathematical Formulation presents the equations related to ramps and reserves.

`max_shutdown_ramp`

> Maximum ramp-down during shutdowns

> **Related Relationship Classes:** unit__from_node and unit__to_node

The definition of the `max_shutdown_ramp` parameter will trigger the creation of the constraint on maximum shutdown ramp. It sets an upper bound on the unit_flow variable for the timestep right before a shutdown.

It can be defined for unit__to_node or unit__from_node relationships, as well as their counterparts for node groups. It will then impose restrictions on the `unit_flow` variables that indicate flows between the two members of the relationship for which the parameter is defined. The parameter is given as a fraction of the unit_capacity parameter. When the parameter is not included, the aforementioned constraint will not be created, which is equivalent to choosing a value of 1.

`max_startup_ramp`

> Maximum ramp-up during startups

> **Related Relationship Classes:** unit__from_node and unit__to_node

The definition of the `max_startup_ramp` parameter will trigger the creation of the Constraint on upward start up ramp_up. It sets an upper bound on the unit_flow variable for the timestep right after a startup.

It can be defined for unit__to_node or unit__from_node relationships, as well as their counterparts for node groups. It will then impose restrictions on the `unit_flow` variables that indicate flows between the two members of the relationship for which the parameter is defined. The parameter is given as a fraction of the unit_capacity parameter. When the parameter is not included, the aforementioned constraint will not be created, which is equivalent to choosing a value of 1.

`max_units_on_coefficient_in_in`

> Optional coefficient for the `units_on` variable impacting the `max_ratio_in_in_unit_flow` constraint.

> **Default value:** 0.0

> **Related Relationship Classes:** unit__node__node

The max_units_on_coefficient_in_in parameter is an optional coefficient in the unit input-input ratio constraint controlled by the max_ratio_in_in_unit_flow parameter. Essentially, it acts as a coefficient for the units_on variable in the constraint, allowing for making the maximum conversion ratio dependent on the amount of online capacity.

Note that there are different parameters depending on the directions of the unit_flow variables being constrained: max_units_on_coefficient_in_out, max_units_on_coefficient_out_in, and max_units_on_coefficient_out_out, all of which apply to their respective constraints. Similarly, there are different parameters for setting minimum or fixed conversion rates, e.g. min_units_on_coefficient_in_in and fix_units_on_coefficient_in_in.

`max_units_on_coefficient_in_out`

> Optional coefficient for the `units_on` variable impacting the `max_ratio_in_out_unit_flow` constraint.

> **Default value:** 0.0

> **Related Relationship Classes:** unit__node__node

The max_units_on_coefficient_in_out parameter is an optional coefficient in the unit input-output ratio constraint controlled by the max_ratio_in_out_unit_flow parameter. Essentially, it acts as a coefficient for the units_on variable in the constraint, allowing for making the maximum conversion ratio dependent on the amount of online capacity.

Note that there are different parameters depending on the directions of the unit_flow variables being constrained: max_units_on_coefficient_in_in, max_units_on_coefficient_out_in, and max_units_on_coefficient_out_out, all of which apply to their respective constraints. Similarly, there are different parameters for setting minimum or fixed conversion rates, e.g. min_units_on_coefficient_in_out and fix_units_on_coefficient_in_out.

`max_units_on_coefficient_out_in`

> Optional coefficient for the `units_on` variable impacting the `max_ratio_out_in_unit_flow` constraint.

> **Default value:** 0.0

> **Related Relationship Classes:** unit__node__node

The max_units_on_coefficient_out_in parameter is an optional coefficient in the unit output-input ratio constraint controlled by the max_ratio_out_in_unit_flow parameter. Essentially, it acts as a coefficient for the units_on in the constraint, allowing for making the maximum conversion ratio dependent on the amount of online capacity.

Note that there are different parameters depending on the directions of the unit_flow being constrained: max_units_on_coefficient_in_in, max_units_on_coefficient_in_out, and max_units_on_coefficient_out_out, all of which apply to their respective constraints. Similarly, there are different parameters for setting minimum or fixed conversion rates, e.g. min_units_on_coefficient_out_in and fix_units_on_coefficient_out_in.

`max_units_on_coefficient_out_out`

> Optional coefficient for the `units_on` variable impacting the `max_ratio_out_out_unit_flow` constraint.

> **Default value:** 0.0

> **Related Relationship Classes:** unit__node__node

The max_units_on_coefficient_out_out parameter is an optional coefficient in the unit output-output ratio constraint controlled by the max_ratio_out_out_unit_flow parameter. Essentially, it acts as a coefficient for the units_on variable in the constraint, allowing for making the maximum conversion ratio dependent on the amount of online capacity.

Note that there are different parameters depending on the directions of the unit_flow variables being constrained: max_units_on_coefficient_in_in, max_units_on_coefficient_out_in, and max_units_on_coefficient_in_out, all of which apply to their respective constraints. Similarly, there are

different parameters for setting minimum or fixed conversion rates, e.g. min_units_on_coefficient_out_out and fix_units_on_coefficient_out_out.

`max_voltage_angle`

> Maximum allowed voltage angle at `node`.

> **Related Object Classes:** node

If a node has a node_voltage_angle variable (see also the parameter has_voltage_angle and this chapter), an upper bound on the voltage angle can be introduced through the max_voltage_angle parameter, which triggers the generation of the maximum node voltage angle constraint.

`min_down_time`

> Minimum downtime of a `unit` after it shuts down.

> **Related Object Classes:** unit

The definition of the `min_down_time` parameter will trigger the creation of the Constraint on minimum down time. It sets a lower bound on the period that a unit has to stay offline after a shutdown.

It can be defined for a unit and will then impose restrictions on the units_on variables that represent the on- or offline status of the unit. The parameter is given as a duration value. When the parameter is not included, the aforementioned constraint will not be created, which is equivalent to choosing a value of 0.

For a more complete description of unit commmitment restrictions, see Unit commitment.

`min_node_pressure`

> Minimum allowed gas pressure at `node`.

> **Related Object Classes:** node

If a node has a node_pressure variable (see also the parameter has_pressure and this chapter), a lower bound on the pressure can be introduced through the min_node_pressure parameter, which triggers the generation of the minimum node pressure constraint.

`min_ratio_in_in_unit_flow`

> Minimum ratio between two `unit_flows` coming into the `unit` from the two `nodes`.

> **Related Relationship Classes:** unit__node__node

The definition of the min_ratio_in_in_unit_flow parameter triggers the generation of the constraint_min_ratio_in_in_unit_flow and sets a lower bound for the ratio between incoming flows of a unit. The parameter is defined on the relationship class unit__node__node, where both nodes (or group of nodes) in this relationship represent `from_nodes`, i.e. the incoming flows to the unit. The ratio parameter is interpreted such that it constrains the ratio of `in1` over `in2`, where `in1` is the unit_flow variable from the first node in the unit__node__node relationship in a left-to-right reading order. This parameter can be useful, for instance if a unit requires a specific commodity mix as a fuel supply.

To enforce e.g. for a unit `u` a minimum share of `0.2` of its incoming flow from the node `supply_fuel_1` compared to its incoming flow from the node group `supply_fuel_2` (consisting of the two nodes `supply_fuel_2_component_a` and `supply_fuel_2_component_b`) the min_ratio_in_in_unit_flow parameter would be set to `0.2` for the relationship `u__supply_fuel_1__supply_fuel_2`.

`min_ratio_in_out_unit_flow`

> Minimum ratio between an incoming `unit_flow` from the first `node` and an outgoing `unit_flow` to the second `node`.

> **Related Relationship Classes:** unit__node__node

The definition of the min_ratio_in_out_unit_flow parameter triggers the generation of the constraint_min_ratio_in_out_unit_flow and enforces a lower bound on the ratio between incoming and outgoing flows of a unit. The parameter is defined on the relationship class unit__node__node, where the first node (or group of nodes, see) in this relationship represents the `from_node`, i.e. the incoming flow to the unit, and the second node (or group of nodes) represents the `to_node` i.e. the outgoing flow from the unit. The ratio parameter is interpreted such that it constrains the ratio of `in` over `out`, where `in` is the unit_flow variable from the first node in the unit__node__node relationship in a left-to-right reading order.

To enforce e.g. a minimum ratio of `1.4` for a unit `u` between its incoming gas flow from the node `ng` and its outgoing flow to the node group `el_heat` (consisting of the two nodes `el` and `heat`), the fix_ratio_in_out_unit_flow parameter would be set to `1.4` for the relationship `u__ng__el_heat`.

`min_ratio_out_in_connection_flow`

> Minimum ratio between the `connection_flow` from the first `node` and the `connection_flow` to the second `node`.

> **Related Relationship Classes:** connection__node__node

The definition of the min_ratio_out_in_connection_flow parameter triggers the generation of the constraint_min_ratio_out_in_connection_flow and sets a lower bound on the ratio between outgoing and incoming flows of a connection. The parameter is defined on the relationship class connection__node__node, where the first node (or group of nodes) in this relationship represents the `to_node`, i.e. the outgoing flow from the connection, and the second node (or group of nodes), represents the `from_node`, i.e. the incoming flows to the connection. The ratio parameter is interpreted such that it constrains the ratio of `out` over `in`, where `out` is the connection_flow variable from the first node in the connection__node__node relationship in a left-to-right reading order.

Note that the ratio can also be defined for connection__node__node relationships, where one or both of the nodes correspond to node groups in order to impose a ratio on aggregated connection flows.

To enforce e.g. a minimum ratio of `0.2` for a connection `conn` between its outgoing electricity flow to node `commodity1` and its incoming flows from the node node `commodity2`, the min_ratio_out_in_connection_flow parameter would be set to `0.8` for the relationship `conn__commodity1__commodity2`.

`min_ratio_out_in_unit_flow`

> Minimum ratio between an outgoing `unit_flow` to the first `node` and an incoming `unit_flow` from the second `node`.

> **Related Relationship Classes:** unit__node__node

The definition of the [min_ratio_out_in_unit_flow] parameter triggers the generation of the constraint_min_ratio_out_in_unit_flow and corresponds to a lower bound of the ratio between out and incoming flows of a unit. The parameter is defined on the relationship class unit__node__node, where

the first node (or group of nodes) in this relationship represents the `to_node`, i.e. the outgoing flow from the unit, and the second node (or group of nodes), represents the `from_node`, i.e. the incoming flows to the unit. The ratio parameter is interpreted such that it constrains the ratio of `out` over `in`, where `out` is the unit_flow variable from the first node in the unit__node__node relationship in a left-to-right reading order.

To enforce e.g. a minimum ratio of `0.8` for a unit `u` between its outgoing flows to the node group `el_heat` (consisting of the two nodes `el` and `heat`) and its incoming gas flow from `ng` the min_ratio_out_in_unit_flow parameter would be set to `0.8` for the relationship `u__el_heat__ng`.

`min_ratio_out_out_unit_flow`

> Minimum ratio between two `unit_flows` going from the `unit` into the two `nodes`.

> **Related Relationship Classes:** unit__node__node

The definition of the min_ratio_out_out_unit_flow parameter triggers the generation of the constraint_min_ratio_out_out_unit_flow and enforces a lower bound on the ratio between outgoing flows of a unit. The parameter is defined on the relationship class unit__node__node, where the nodes (or group of nodes) in this relationship represent the `to_node`'s', i.e. outgoing flow from the unit. The ratio parameter is interpreted such that it constrains the ratio of `out1` over `out2`, where `out1` is the unit_flow variable from the first node in the unit__node__node relationship in a left-to-right reading order.

To enforce a minimum ratio between two products of a unit `u`, e.g. setting the minimum share of produced electricity flowing to node `el` to `0.4` of the production of heat flowing to node `heat`, the fix_ratio_out_out_unit_flow parameter would be set to `0.4` for the relationship `u__el__heat`.

`min_res_shutdown_ramp`

> Minimum non-spinning reserve ramp-down for online units providing reserves during shut-downs

> **Related Relationship Classes:** unit__from_node and unit__to_node

A unit can provide spinning and nonspinning reserves to a reserve node. These reserves can be either upward_reserve or downward_reserve. Nonspinning downward reserves are provided by contracted units holding available to shutdown to a downward_reserve node. If a unit is scheduled to provide

nonspinning reserve, a limit on the minimum amount of reserves provided can be imposed by defining the parameter min_res_shutdown_ramp on a unit__to_node relationship, which triggers the constraint on minimum downward nonspinning reserve provision. The parameter min_res_shutdown_ramp is given as a fraction of the unit_capacity of the corresponding unit__to_node relationship.

Note that to include the provision of nonspinning downward reserves, the parameter max_res_shutdown_ramp needs to be defined on the corresponding unit__to_node relationship, which triggers the generation of the variables nonspin_units_shut_down and nonspin_ramp_down_unit_flow.

A detailed description of the usage of ramps and reserves is given in the chapter Ramping and Reserves. The chapter Ramping and reserve constraints in the Mathematical Formulation presents the equations related to ramps and reserves.

`min_res_startup_ramp`

> Minimum non-spinning reserve ramp-up for offline units scheduled for reserve provision

> **Related Relationship Classes:** unit__from_node and unit__to_node

A unit can provide spinning and nonspinning reserves to a reserve node. These reserves can be either upward_reserve or downward_reserve. Nonspinning upward reserves are provided to an upward_reserve node by contracted offline units holding available to startup. If a unit is scheduled to provide nonspinning reserve, a limit on the minimum amount of reserves provided can be imposed by defining the parameter min_res_startup_ramp on a unit__to_node relationship, which triggers the constraint on minimum upward nonspinning reserve provision. The parameter min_res_startup_ramp is given as a fraction of the unit_capacity of the corresponding unit__to_node relationship.

Note that to include the provision of nonspinning upward reserves, the parameter max_res_startup_ramp needs to be defined on the corresponding unit__to_node relationship, which triggers the generation of the variables nonspin_units_started_up and nonspin_ramp_up_unit_flow.

A detailed description of the usage of ramps and reserves is given in the chapter Ramping and Reserves. The chapter Ramping and reserve constraints in the Mathematical Formulation presents the equations related to ramps and reserves.

`min_shutdown_ramp`

> Minimum ramp-up during startups

> **Related Relationship Classes:** unit__from_node and unit__to_node

The definition of the `min_shutdown_ramp` parameter will trigger the creation of the constraint on minimum shutdown ramp. It sets a lower bound on the unit_flow variable for the timestep right before a shutdown.

It can be defined for unit__to_node or unit__from_node relationships, as well as their counterparts for node groups. It will then impose restrictions on the `unit_flow` variables that indicate flows between the two members of the relationship for which the parameter is defined. The parameter is given as a fraction of the unit_capacity parameter. When the parameter is not included, the aforementioned constraint will not be created, which is equivalent to choosing a value of 0.

`min_startup_ramp`

> Minimum ramp-up during startups

> **Related Relationship Classes:** unit__from_node and unit__to_node

The definition of the `min_startup_ramp` parameter will trigger the creation of the constraint on minimum startup ramp. It sets a lower bound on the unit_flow variable for the timestep right after a startup.

It can be defined for unit__to_node or unit__from_node relationships, as well as their counterparts for node groups. It will then impose restrictions on the `unit_flow` variables that indicate flows between the two members of the relationship for which the parameter is defined. The parameter is given as a fraction of the unit_capacity parameter. When the parameter is not included, the aforementioned constraint will not be created, which is equivalent to choosing a value of 0.

`min_units_on_coefficient_in_in`

> Optional coefficient for the `units_on` variable impacting the `min_ratio_in_in_unit_flow` constraint.

> **Default value:** 0.0

> **Related Relationship Classes:** unit__node__node

The min_units_on_coefficient_in_in parameter is an optional coefficient in the unit input-input ratio constraint controlled by the min_ratio_in_in_unit_flow parameter. Essentially, it acts as a coefficient for the units_on variable in the constraint, allowing for making the minimum conversion ratio dependent on the amount of online capacity.

Note that there are different parameters depending on the directions of the unit_flow variables being constrained: min_units_on_coefficient_in_out, min_units_on_coefficient_out_in, and min_units_on_coefficient_out_out, all of which apply to their respective constraints. Similarly, there are different parameters for setting maximum or fixed conversion rates, e.g. max_units_on_coefficient_in_in and fix_units_on_coefficient_in_in.

`min_units_on_coefficient_in_out`

> Optional coefficient for the `units_on` variable impacting the `min_ratio_in_out_unit_flow` constraint.

> **Default value:** 0.0

> **Related Relationship Classes:** unit__node__node

The min_units_on_coefficient_in_out parameter is an optional coefficient in the unit input-output ratio constraint controlled by the min_ratio_in_out_unit_flow parameter. Essentially, it acts as a coefficient for the units_on variable in the constraint, allowing for making the minimum conversion ratio dependent on the amount of online capacity.

Note that there are different parameters depending on the directions of the unit_flow variables being constrained: min_units_on_coefficient_in_in, min_units_on_coefficient_out_in, and min_units_on_coefficient_out_out, all of which apply to their respective constraints. Similarly, there are different parameters for setting maximum or fixed conversion rates, e.g. max_units_on_coefficient_in_out and fix_units_on_coefficient_in_out.

`min_units_on_coefficient_out_in`

> Optional coefficient for the `units_on` variable impacting the `min_ratio_out_in_unit_flow` constraint.

> **Default value:** 0.0

> **Related Relationship Classes:** unit__node__node

The min_units_on_coefficient_out_in parameter is an optional coefficient in the unit output-input ratio constraint controlled by the min_ratio_out_in_unit_flow parameter. Essentially, it acts as a coefficient for the units_on variable in the constraint, allowing for making the minimum conversion ratio dependent on the amount of online capacity.

Note that there are different parameters depending on the directions of the unit_flow variables being constrained: min_units_on_coefficient_in_in, min_units_on_coefficient_in_out, and min_units_on_coefficient_out_out, all of which apply to their respective constraints. Similarly, there are different parameters for setting maximum or fixed conversion rates, e.g. max_units_on_coefficient_out_in and fix_units_on_coefficient_out_in.

`min_units_on_coefficient_out_out`

> Optional coefficient for the `units_on` variable impacting the `min_ratio_out_out_unit_flow` constraint.

> **Default value:** 0.0

> **Related Relationship Classes:** unit__node__node

The min_units_on_coefficient_out_out parameter is an optional coefficient in the unit output-output ratio constraint controlled by the min_ratio_out_out_unit_flow parameter. Essentially, it acts as a coefficient for the units_on variable in the constraint, allowing for making the minimum conversion ratio dependent on the amount of online capacity.

Note that there are different parameters depending on the directions of the unit_flow variables being constrained: min_units_on_coefficient_in_in, min_units_on_coefficient_in_out, and min_units_on_coefficient_out_in, all of which apply to their respective constraints. Similarly, there are different parameters for setting maximum or fixed conversion rates, e.g. max_units_on_coefficient_out_out and fix_units_on_coefficient_out_out.

`min_up_time`

> Minimum uptime of a `unit` after it starts up.

> **Related Object Classes:** unit

The definition of the `min_up_time` parameter will trigger the creation of the Constraint on minimum up time. It sets a lower bound on the period that a unit has to stay online after a startup.

It can be defined for a unit and will then impose restrictions on the units_on variables that represent the on- or offline status of the unit. The parameter is given as a duration value. When the parameter is not included, the aforementioned constraint will not be created, which is equivalent to choosing a value of 0.

For a more complete description of unit commmitment restrictions, see Unit commitment.

`min_voltage_angle`

> Minimum allowed voltage angle at `node`.

> **Related Object Classes:** node

If a node has a node_voltage_angle variable (see also the parameter has_voltage_angle and this chapter), a lower bound on the pressure can be introduced through the min_voltage_angle parameter, which triggers the generation of the minimum node voltage angle constraint.

`minimum_operating_point`

> Minimum level for the `unit_flow` relative to the `units_on` online capacity.

> **Related Relationship Classes:** unit__from_node and unit__to_node

The definition of the `minimum_operating_point` parameter will trigger the creation of the Constraint on minimum operating point. It sets a lower bound on the value of the unit_flow variable for a unit that is online.

It can be defined for unit__to_node or unit__from_node relationships, as well as their counterparts for node groups. It will then impose restrictions on the `unit_flow` variables that indicate flows between

the two members of the relationship for which the parameter is defined. The parameter is given as a fraction of the unit_capacity parameter. When the parameter is not included, the aforementioned constraint will not be created, which is equivalent to choosing a value of 0.

`minimum_reserve_activation_time`

> Duration a certain reserve product needs to be online/available

> **Related Object Classes:** node

The parameter minimum_reserve_activation_time is the duration a reserve product needs to be online, before it can be replaced by another (slower) reserve product.

In SpineOpt, the parameter is used to model reserve provision through storages. If a storage provides reserves to a reserve node (see also is_reserve_node) one needs to ensure that the node state is sufficiently high to provide these scheduled reserves as least for the duration of the minimum_reserve_activation_time. The constraint on the minimum node state with reserve provision is triggered by the existence of the minimum_reserve_activation_time. See also Ramping and Reserves

`model_end`

> Defines the last timestamp to be modelled. Rolling optimization terminates after passing this point.

> **Default value:** Dict{String, Any}("data" => "2000-01-02T00:00:00", "type" => "date_time")

> **Related Object Classes:** model

Together with the model_start parameter, it is used to define the temporal horizon of the model. In case of a single solve optimization, the parameter marks the end of the last timestep that is possibly part of the optimization. Note that it poses an upper bound, and that the optimization does not necessarily include this timestamp when the block_end parameters are more stringent.

In case of a rolling horizon optimization, it will tell to the model to stop rolling forward once an optimization has been performed for which the result of the indicated timestamp has been kept in the final results. For example, assume that a `model_end` value of `2030-01-01T05:00:00` has been chosen, a block_end of `3h`, and a roll_forward of `2h`. The roll_forward parameter indicates here that the results of

the first two hours of each optimization window are kept as final, therefore the last optimization window will span the timeframe `[2030-01-01T04:00:00 - 2030-01-01T06:00:00]`.

A DateTime value should be chosen for this parameter.

`model_start`

> Defines the first timestamp to be modelled. Relative `temporal_blocks` refer to this value for their start and end.

> **Default value:** Dict{String, Any}("data" => "2000-01-01T00:00:00", "type" => "date_time")

> **Related Object Classes:** model

Together with the model_end parameter, it is used to define the temporal horizon of the model. For a single solve optimization, it marks the timestamp from which the relative offset in a temporal_block is defined by the block_start parameter. In the rolling optimization framework, it does this for the first optimization window.

A DateTime value should be chosen for this parameter.

`model_type`

> Used to identify model objects as relating to the master problem or operational sub problems (default)

> **Default value:** spineopt_operations

> **Uses Parameter Value Lists:** model_type_list

> **Related Object Classes:** model

This parameter is used, generally, to control model dependent functionality and specify model-level parameters for different models. Currently, the main use is to identify the model objects that represent

the master and operational sub problems within a decomposed investment problem structure. To trigger the decomposed structure, a model object with `model_type=:spineopt_master` must exist and another with `model_type=:spineopt_operations` must also be present. To deactivate the decomposition functionality, the `model_type` of the master problem can be set to `:spineopt_other`.

See also Decomposition.

`nodal_balance_sense`

> A selector for `nodal_balance` constraint sense.

> **Default value:** ==

> **Uses Parameter Value Lists:** constraint_sense_list

> **Related Object Classes:** node

nodal_balance_sense determines whether or not a node is able to naturally consume or produce energy. The default value, `==`, means that the node is unable to do any of that, and thus it needs to be perfectly balanced. The vale `>=` means that the node is a *sink*, that is, it can *consume* any amounts of energy. The value `<=` means that the node is a *source*, that is, it can *produce* any amounts of energy.

`node_opf_type`

> A selector for the reference `node` (slack bus) when PTDF-based DC load-flow is enabled.

> **Default value:** node_opf_type_normal

> **Uses Parameter Value Lists:** node_opf_type_list

> **Related Object Classes:** node

Used to identify the reference node (or slack bus) when ptdf based dc load flow is enabled (commodity_physics set to commodity_physics_ptdf or commodity_physics_lodf. To identify the reference node, set `node_opf_type` = `:node_opf_type_reference`

See also powerflow.

node_slack_penalty

> A penalty cost for `node_slack_pos` and `node_slack_neg` variables. The slack variables won't be included in the model unless there's a cost defined for them.

> **Related Object Classes:** node

`node_slack_penalty` triggers the creation of node slack variables, `node_slack_pos` and `node_slack_neg`. This allows the model to violate the node_balance constraint with these violations penalised in the objective function with a coefficient equal to `node_slack_penalty`. If `node_slack_penalty` = 0 the slack variables are created and violations are unpenalised. If set to none or undefined, the variables are not created and violation of the node_balance constraint is not possible.

node_state_cap

> The maximum permitted value for a `node_state` variable.

> **Related Object Classes:** node

The node_state_cap parameter represents the maximum allowed value for the `node_state` variable. Note that in order for a node to have a `node_state` variable in the first place, the has_state parameter must be set to `true`. However, if the node has storage investments enabled using the candidate_storages parameter, the node_state_cap parameter acts as a coefficient for the `storages_invested_available` variable. Essentially, with investments, the node_state_cap parameter represents *storage capacity per storage investment*.

node_state_coefficient

> Coefficient of the specified node's state variable in the specified unit constraint.

> **Default value:** 0.0

> **Related Relationship Classes:** node__unit_constraint

The node_state_coefficient is an optional parameter that can be used to include the node_state variable of a node in a unit_constraint via the node__unit_constraint relationship. Essentially, node_state_coefficient appears as a coefficient for the node_state variable of the node in the unit constraint.

`node_state_min`

> The minimum permitted value for a `node_state` variable.

> **Default value:** 0.0

> **Related Object Classes:** node

The node_state_min parameter sets the lower bound for the node_state variable, if one has been enabled by the has_state parameter. For reserve nodes with minimum_reserve_activation_time, the node_state_min is considered also via a special constraint.

`number_of_units`

> Denotes the number of 'sub units' aggregated to form the modelled `unit`.

> **Default value:** 1.0

> **Related Object Classes:** unit

Defines how many members a certain unit object represents. Typically this parameter takes a binary (UC) or integer (clustered UC) value. Together with the unit_availability_factor, this will determine the maximum number of members that can be online at any given time. (Thus restricting the units_on

variable). It is possible to allow the model to increase the `number_of_units` itself, through Investment Optimization

The default value for this parameter is 1.

`online_variable_type`

> A selector for how the `units_on` variable is represented within the model.

> **Default value:** unit_online_variable_type_linear

> **Uses Parameter Value Lists:** unit_online_variable_type_list

> **Related Object Classes:** unit

`online_variable_type` is a method parameter closely related to the number_of_units and can take the values "unit_online_variable_type_binary", "unit_online_variable_type_integer", "unit_online_variable_type_linear". If the binary value is chosen, the units status is modelled as a binary (classic UC). For clustered unit commitment units, the integer type is applicable. Note that if the parameter is not defined, the default will be linear. If the units status is not crucial, this can reduce the computational burden.

`operating_points`

> Decomposes the flow variable into a number of separate operating segment variables. Used to in conjunction with `unit_incremental_heat_rate` and/or `unit_constraints`

> **Related Relationship Classes:** unit__from_node and unit__to_node

If `operating_points` is defined as an array type on a certain `unit__to_node` or `unit__from_node` flow, the corresponding `unit_flow` flow variable is decomposed into a number of sub operating segment variables, `unit_flow_op` one for each operating segment, with an additional index, `i` to reference the specific operating segment. Each value in the array represents the upper bound of the operating segment, normalized on `unit_capacity` for the corresponding `unit__to_node` or `unit__from_node` flow. `operating_points` is used in conjunction with unit_incremental_heat_rate

where the array dimension must match and is used to define the normalized operating point bounds for the corresponding incremental heat rate. `operating_points` is also used in conjunction with unit_constraint where the array dimension must match any corresponding piecewise linear unit_flow_coefficient. Here `operating_points` is used also to define the normalized operating point bounds for the corresponding unit_flow_coefficients.

Note that `operating_points` is defined on a capacity-normalized basis and the values represent the upper bound of the corresponding operating segment variable. So if `operating_points` is specified as [0.5, 1], this creates two operating segments, one from zero to 50% of the corresponding unit_capacity and a second from 50% to 100% of the corresponding unit_capacity.

`output_db_url`

> Database url for SpineOpt output.

> **Related Object Classes:** report

The output_db_url parameter is the url of the databse to write the results of the model run. It overrides the value of the second argument passed to `run_spineopt`.

`ramp_down_cost`

> Costs of ramping down

> **Related Relationship Classes:** unit__from_node and unit__to_node

By defining the ramp_down_cost parameter for a specific unit__to_node or unit__from_node relationship, a cost term will be added to the objective function whenever the unit ramps down its activity (i.e., when the ramp_down_unit_flow is nonzero) over the course of its operational dispatch during the current optimization window.

`ramp_down_limit`

> Limit the maximum ramp-down rate of an online unit, given as a fraction of the unit*capacity. [ramp*down_limit] = %/t, e.g. 0.2/h

> **Related Relationship Classes:** unit__from_node and unit__to_node

The definition of the `ramp_down_limit` parameter will trigger the creation of the Constraint on spinning downward ramps. It will limit the maximum decrease in the unit_flow variable between two consecutive timesteps for which the unit is online.

It can be defined for unit__to_node or unit__from_node relationships, as well as their counterparts for node groups. It will then impose restrictions on the `unit_flow` variables that indicate flows between the two members of the relationship for which the parameter is defined. The parameter is given as a fraction of the unit_capacity parameter. When the parameter is not included, the aforementioned constraint will not be created, which is equivalent to choosing a value of 1.

`ramp_up_cost`

> Costs of ramping up

> **Related Relationship Classes:** unit__from_node and unit__to_node

By defining the ramp_up_cost parameter for a specific unit__to_node or unit__from_node relationship, a cost term will be added to the objective function whenever the unit ramps up its activity (i.e., when the ramp_up_unit_flow is nonzero) over the course of its operational dispatch during the current optimization window.

`ramp_up_limit`

> Limit the maximum ramp-up rate of an online unit, given as a fraction of the unit*capacity.*
> *[ramp*up_limit] = %/t, e.g. 0.2/h

> **Related Relationship Classes:** unit__from_node and unit__to_node

The definition of the `ramp_up_limit` parameter will trigger the creation of the Constraint on spinning upwards ramp_up. It limits the maximum increase in the unit_flow variable between two consecutive timesteps for which the unit is online.

It can be defined for unit__to_node or unit__from_node relationships, as well as their counterparts for node groups. It will then impose restrictions on the `unit_flow` variables that indicate flows between the two members of the relationship for which the parameter is defined. The parameter is given as a

fraction of the unit_capacity parameter. When the parameter is not included, the aforementioned constraint will not be created, which is equivalent to choosing a value of 1.

For a more complete description of how ramping restrictions can be implemented, see Ramping and Reserves.

representative_periods_mapping

> Mapping from real timelices to their corresponding representative days

> **Related Object Classes:** temporal_block

For representative periods with seasonal storages, SpineOpt.jl can be interlinked with the package SpinePeriods.jl. SpinePeriods.jl provides the representative_periods_mapping parameter, which maps each non-representative period of the whole optimization window to its representative temporal_block. The map is organized as timeseries (indicating the start of each the non-representative period) with the names of the representative temporal_blocks as entries.

reserve_procurement_cost

> Procurement cost for reserves

> **Related Relationship Classes:** unit__from_node and unit__to_node

By defining the reserve_procurement_cost parameter for a specific unit__to_node or unit__from_node relationship, a cost term will be added to the objective function whenever that unit is used over the course of the operational dispatch during the current optimization window.

resolution

> Temporal resolution of the `temporal_block`. Essentially, divides the period between `block_start` and `block_end` into `TimeSlices` with the input `resolution`.

> **Default value:** Dict{String, Any}("data" => "1h", "type" => "duration")

> **Related Object Classes:** temporal_block

This parameter specifies the resolution of the temporal block, or in other words: the length of the timesteps used in the optimization run. Generally speaking, variables and constraints are generated for each timestep of an optimization. For example, the nodal balance constraint must hold for each timestep.

An array of duration values can be used to have a resolution that varies with time itself. It can for example be used when uncertainty in one of the inputs rises as the optimization moves away from the model start. Think of a forecast of for instance wind power generation, which might be available in quarter hourly detail for one day in the future, and in hourly detail for the next two days. It is possible to take a quarter hourly resolution for the full horizon of three days. However, by lowering the temporal resolution after the first day, the computational burden is lowered substantially.

`right_hand_side`

> The right-hand side, constant term in a `unit_constraint`. Can be time-dependent and used e.g. for complicated efficiency approximations.

> **Default value:** 0.0

> **Related Object Classes:** unit_constraint

Used to specify the right-hand-side, constant term in a unit_constraint. See also unit_constraint.

`roll_forward`

> Defines how much the model moves ahead in time between solves in a rolling optimization. Without this parameter, everything is solved in as a single optimization.

> **Related Object Classes:** model

This parameter defines how much the optimization window rolls forward in a rolling horizon optimization and should be expressed as a duration. In a rolling horizon optimization, a (small) part of the model is optimized at each iteration, after which the window rolls forward to optimize a different

part. Overlap between consecutive optimization windows is possible. In the practical approaches presented in Temporal Framework, the rolling window optimization will be explained in more detail. The default value of this parameter is the entire model time horizon, which leads to a single optimization for the entire time horizon.

`shut_down_cost`

> Costs of shutting down a 'sub unit', e.g. EUR/shutdown.

> **Related Object Classes:** unit

By defining the shut_down_cost parameter for a specific `unit`, a cost term will be added to the objective function whenever this unit shuts down over the course of its operational dispatch during the current optimization window.

`start_up_cost`

> Costs of starting up a 'sub unit', e.g. EUR/startup.

> **Related Object Classes:** unit

By defining the start_up_cost parameter for a specific `unit`, a cost term will be added to the objective function whenever this unit starts up over the course of its operational dispatch during the current optimization window.

`state_coeff`

> Represents the `commodity` content of a `node_state` variable in respect to the `unit_flow` and `connection_flow` variables. Essentially, acts as a coefficient on the `node_state` variable in the `:node_injection` constraint.

> **Default value:** 0.0

> **Related Object Classes:** node

The state_coeff parameter acts as a coefficient for the `node_state` variable in the node injection constraint. Essentially, it tells how the `node_state` variable should be treated in relation to the commodity flows and demand, and can be used for e.g. scaling or unit conversions. For most use-cases a state_coeff parameter value of `1.0` should suffice, e.g. having a MWh storage connected to MW flows in a model with hour as the basic unit of time.

Note that in order for the state_coeff parameter to have an impact, the node must first have a `node_state` variable to begin with, defined using the has_state parameter. By default, the state_coeff is set to zero as a precaution, so that the user always has to set its value explicitly for it to have an impact on the model.

`stochastic_scenario_end`

> A `Duration` for when a `stochastic_scenario` ends and its `child_stochastic_scenarios` start. Values are interpreted relative to the start of the current solve, and if no value is given, the `stochastic_scenario` is assumed to continue indefinitely.

> **Related Relationship Classes:** stochastic_structure__stochastic_scenario

The stochastic_scenario_end is a `Duration`-type parameter, defining when a stochastic_scenario ends relative to the start of the current optimization. As it is a parameter for the stochastic_structure__stochastic_scenario relationship, different stochastic_structures can have different values for the same stochastic_scenario, making it possible to define slightly different stochastic_structures using the same stochastic_scenarios. See the Stochastic Framework section for more information about how different stochastic_structures interact in *SpineOpt.jl*.

When a stochastic_scenario ends at the point in time defined by the stochastic_scenario_end parameter, it spawns its children according to the parent_stochastic_scenario__child_stochastic_scenario relationship. Note that the children will be inherently assumed to belong to the same stochastic_structure their parent belonged to, even without explicit stochastic_structure__stochastic_scenario relationships! Thus, you might need to define the weight_relative_to_parents parameter for the children.

If no stochastic_scenario_end is defined, the stochastic_scenario is assumed to go on indefinitely.

`storage_investment_cost`

> Determines the investment cost per unit state_cap over the investment life of a storage

> **Related Object Classes:** node

By defining the storage_investment_cost parameter for a specific `node`, a cost term will be added to the objective function whenever a storage investment is made during the current optimization window.

`storage_investment_lifetime`

> Minimum lifetime for storage investment decisions.

> **Related Object Classes:** node

Duration parameter that determines the minimum duration of storage investment decisions. Once a storage has been invested-in, it must remain invested-in for `storage_investment_lifetime`. Note that `storage_investment_lifetime` is a dynamic parameter that will impact the amount of solution history that must remain available to the optimisation in each step - this may impact performance.

See also Investment Optimization and candidate_storages

`storage_investment_variable_type`

> Determines whether the storage investment variable is continuous (usually representing capacity) or integer (representing discrete units invested)

> **Default value:** variable_type_integer

> **Uses Parameter Value Lists:** variable_type_list

> **Related Object Classes:** node

Within an investments problem `storage_investment_variable_type` determines the storage investment decision variable type. Since a node's `node_state` will be limited to the product of the investment variable and the corresponding node_state_cap and since candidate_storages represents the upper bound of the storage investment decision variable, `storage_investment_variable_type` thus determines what the investment decision represents. If storage_investment_variable_type is integer or binary, then candidate_storages represents the maximum number of discrete storages that may be invested-in. If storage_investment_variable_type is continuous, `candidate_storages` is more analagous to a capacity with node_state_cap being analagous to a scaling parameter. For example, if `storage_investment_variable_type` = `integer`, `candidate_storages` = 4 and `node_state_cap` = 1000 MWh, then the investment decision is how many 1000h MW storages to build. If `storage_investment_variable_type` = continuous, `candidate_storages` = 1000 and `node_state_cap` = 1 MWh, then the investment decision is how much storage capacity to build. Finally, if `storage_investment_variable_type` = `integer`, `candidate_storages` = 10 and `node_state_cap` = 100 MWh, then the investment decision is how many 100MWh storage blocks to build.

See also Investment Optimization and candidate_storages.

`tax_in_unit_flow`

> Tax costs for incoming `unit_flows` on this `node`. E.g. EUR/MWh.

> **Default value:** 0.0

> **Related Object Classes:** node

By defining the tax_in_unit_flow parameter for a specific `node`, a cost term will be added to the objective function to account the taxes associated with all unit_flow variables with direction `to_node` over the course of the operational dispatch during the current optimization window.

`tax_net_unit_flow`

> Tax costs for net incoming and outgoing `unit_flows` on this `node`. Incoming flows accrue positive net taxes, and outgoing flows accrue negative net taxes.

> **Default value:** 0.0

> **Related Object Classes:** node

By defining the tax_net_unit_flow parameter for a specific node, a cost term will be added to the objective function to account the taxes associated with the net total of all unit_flow variables with the direction `to_node` for this specific node minus all unit_flow variables with direction `from_node`.

`tax_out_unit_flow`

> Tax costs for outgoing `unit_flows` from this `node`. E.g. EUR/MWh.

> **Default value:** 0.0

> **Related Object Classes:** node

By defining the tax_out_unit_flow parameter for a specific `node`, a cost term will be added to the objective function to account the taxes associated with all unit_flow variables with direction `from_node` over the course of the operational dispatch during the current optimization window.

`unit_availability_factor`

> Availability of the `unit`, acting as a multiplier on its `unit_capacity`. Typically between 0-1.

> **Default value:** 1.0

> **Related Object Classes:** unit

To indicate that a unit is only available to a certain extent or at certain times of the optimization, the unit_availability_factor can be used. A typical use case could be an availability timeseries for a variable renewable energy source. By default the availability factor is set to `1`. The availability is, among others, used in the constraint_units_available.

`unit_capacity`

> Maximum `unit_flow` capacity of a single 'sub_unit' of the `unit`.

> **Related Relationship Classes:** unit__from_node and unit__to_node

To set an upper bound on the commodity flow of a unit in a certain direction, the unit_capacity constraint needs to be defined on a unit__to_node or unit__from_node relationship. By defining the parameter, the unit_flow variables to or from a node or a group of nodes will be constrained by the capacity constraint.

Note that if the unit_capacity parameter is defined on a node group, the sum of all unit_flows within the specified node group will be constrained by the unit_capacity.

`unit_conv_cap_to_flow`

> Optional coefficient for `unit_capacity` unit conversions in the case the `unit_capacity` value is incompatible with the desired `unit_flow` units.

> **Default value:** 1.0

> **Related Relationship Classes:** unit__from_node and unit__to_node

The `unit_conv_cap_to_flow`, as defined for a unit__to_node or unit__from_node, allows the user to align between unit_flow variables and the unit_capacity parameter, which may be expressed in different units. An example would be when the `unit_capacity` is expressed in GWh, while the demand on the node is expressed in MWh. In that case, a `unit_conv_cap_to_flow` parameter of 1000 would be applicable.

`unit_flow_coefficient`

> Coefficient of a `unit_flow` variable for a custom `unit_constraint`.

> **Default value:** 0.0

> **Related Relationship Classes:** unit__from_node__unit_constraint and unit__to_node__unit_constraint

The unit_flow_coefficient is an optional parameter that can be used to include the unit_flow or unit_flow_op variables from or to a node in a unit_constraint via the unit__from_node__unit_constraint and unit__to_node__unit_constraint relationships. Essentially, unit_flow_coefficient appears as a coefficient for the unit_flow and unit_flow_op variables from or to the node in the unit constraint.

Note that the unit_flow_op variables are a bit of a special case, defined using the operating_points parameter.

`unit_idle_heat_rate`

> Flow from node1 per unit time and per `units_on` that results in no additional flow to node2

> **Default value:** 0.0

> **Related Relationship Classes:** unit__node__node

Used to implement the no-load or idle heat rate of a unit. This is the y-axis offset of the heat rate function and is the fuel consumed per unit time when a unit is online and that results in no additional output. This is defined on the `unit__node__node` relationship and it is assumed that the input flow from node 1 represents fuel consumption and the output flow to node 2 is the elecrical output. While the units depend on the data, `unit_idle_heat_rate` is generally expressed in GJ/hr. Used in conjunction with unit_incremental_heat_rate. `unit_idle_heat_rate` is only currently considered if unit_incremental_heat_rate is specified. A trivial unit_incremental_heat_rate of zero can be defined if there is no incremental heat rate.

`unit_incremental_heat_rate`

> Standard piecewise incremental heat rate where node1 is assumed to be the fuel and node2 is assumed to be electriciy. Assumed monotonically increasing. Array type or single coefficient where the number of coefficients must match the dimensions of `unit_operating_points`

> **Related Relationship Classes:** unit__node__node

Used to implement simple or piecewise linear incremental heat rate functions. Used in the constraint unit_pw_heat_rate - the input fuel flow at node 1 is the sum of the electrical MW output at node 2 times the incremental heat rate over all heat rate segments, plus the unit_idle_heat_rate. The units are detmerined by the data, but generally, incremental heat rates are given in GJ/MWh. Note that the formulation assumes a convex, monitonically increasing heat rate function. The formulation relies on optimality to load the heat rate segments in the correct order and no additional integer variables are created to enforce the correct loading order. The heat rate segment MW operating points are defined by operating_points.

To implement a simple incremental heat rate function, `unit_incremental_heat_rate` should be given as a simple scalar representing the incremental heat rate over the entire operating range of the unit. To implement a piecewise linear heat rate function, `unit_incremental_heat_rate` should be specified as an array type. It is then used in conjunction with the unit parameter operating_points which should also be defined as an array type of equal dimension. When defined as an array type `unit_incremental_heat_rate`[i] is the effective incremental heat rate between operating_points [i-1] (or zero if i=1) and operating_points[i]. Note that operating_points is defined on a capacity-normalized basis so if operating_points is specified as [0.5, 1], this creates two operating segments, one from zero to 50% of the corresponding unit_capacity and a second from 50% to 100% of the corresponding unit_capacity.

`unit_investment_cost`

> Investment cost per 'sub unit' built.

> **Related Object Classes:** unit

By defining the unit_investment_cost parameter for a specific `unit`, a cost term will be added to the objective function whenever a unit investment is made during the current optimization window.

`unit_investment_lifetime`

> Minimum lifetime for unit investment decisions.

> **Related Object Classes:** unit

Duration parameter that determines the minimum duration of `unit` investment decisions. Once a `unit` has been invested-in, it must remain invested-in for `unit_investment_lifetime`. Note that

`unit_investment_lifetime` is a dynamic parameter that will impact the amount of solution history that must remain available to the optimisation in each step - this may impact performance.

See also Investment Optimization and candidate_units

unit_investment_variable_type

> Determines whether investment variable is integer or continuous.

> **Default value:** unit_investment_variable_type_continuous

> **Uses Parameter Value Lists:** unit_investment_variable_type_list

> **Related Object Classes:** unit

Within an investments problem `unit_investment_variable_type` determines the unit investment decision variable type. Since the `unit_flow`s will be limited to the product of the investment variable and the corresponding unit_capacity for each `unit_flow` and since candidate_units represents the upper bound of the investment decision variable, `unit_investment_variable_type` thus determines what the investment decision represents. If unit_investment_variable_type is integer or binary, then candidate_units represents the maximum number of discrete units that may be invested. If unit_investment_variable_type is continuous, `candidate_units` is more analagous to a capacity with unit_capacity being analagous to a scaling parameter. For example, if `unit_investment_variable_type` = `integer`, `candidate_units` = 4 and `unit_capacity` for a particular `unit_flow` = 400 MW, then the investment decision is how many 400 MW units to build. If `unit_investment_variable_type` = continuous, `candidate_units` = 400 and `unit_capacity` for a particular `unit_flow` = 1 MW, then the investment decision is how much capacity if this particular unit to build. Finally, if `unit_investment_variable_type` = `integer`, `candidate_units` = 10 and `unit_capacity` for a particular `unit_flow` = 50 MW, then the investment decision is many 50MW blocks of capacity of this particular unit to build.

See also Investment Optimization and candidate_units

unit_start_flow

> Flow from node1 that is incurred when a unit is started up.

> **Default value:** 0.0

> **Related Relationship Classes:** unit__node__node

Used to implement unit startup fuel consumption where node 1 is assumed to be input fuel and node 2 is assumed to be output elecrical energy. This is a flow from node 1 that is incurred when the value of the variable units*started*up is 1 in the corresponding time period. This flow does not result in additional output flow at node 2. Used in conjunction with unit_incremental_heat_rate. `unit_start_flow` is only currently considered if unit_incremental_heat_rate is specified. A trivial unit_incremental_heat_rate of zero can be defined if there is no incremental heat rate.

`units_on_coefficient`

> Coefficient of a `units_on` variable for a custom `unit_constraint`.

> **Default value:** 0.0

> **Related Relationship Classes:** unit__unit_constraint

The units_on_coefficient is an optional parameter that can be used to include the units_on variable of a unit in a unit_constraint via the unit__unit_constraint relationship. Essentially, units_on_coefficient appears as a coefficient for the units_on variable of the unit in the unit constraint.

`units_started_up_coefficient`

> Coefficient of a `units_started_up` variable for a custom `unit_constraint`.

> **Default value:** 0.0

> **Related Relationship Classes:** unit__unit_constraint

The units_started_up_coefficient is an optional parameter that can be used to include the units_started_up variable of a unit in a unit_constraint via the unit__unit_constraint relationship. Essentially, units_started_up_coefficient appears as a coefficient for the units_started_up variable of the unit in the unit constraint.

`upward_reserve`

> Identifier for `nodes` providing upward reserves

> **Default value:** false

> **Related Object Classes:** node

If a node has a `true` is_reserve_node parameter, it will be treated as a reserve node in the model. To define whether the node corresponds to an upward or downward reserve commodity, the upward_reserve or the downward_reserve parameter needs to be set to `true`, respectively.

`vom_cost`

> Variable operating costs of a `unit_flow` variable. E.g. EUR/MWh.

> **Related Relationship Classes:** unit__from_node and unit__to_node

By defining the vom_cost parameter for a specific `unit`, `node`, and `direction`, a cost term will be added to the objective function to account for the variable operation and maintenance costs associated with that unit over the course of its operational dispatch during the current optimization window.

`weight`

> Weighting factor of the temporal block associated with the objective function

> **Default value:** 1.0

> **Related Object Classes:** temporal_block

The `weight` variable, defined for a temporal_block object can be used to assign different weights to different temporal periods that are modeled. It basically determines how important a certain temporal period is in the total cost, as it enters the Objective function. The main use of this parameter is for representative periods, where each representative period represents a specific fraction of a year or so.

`weight_relative_to_parents`

> The weight of the `stochastic_scenario` in the objective function relative to its parents.

> **Default value:** 1.0

> **Related Relationship Classes:** stochastic_structure__stochastic_scenario

The weight_relative_to_parents parameter defines how much weight the stochastic_scenario gets in the Objective function. As a stochastic_structure__stochastic_scenario relationship parameter, different stochastic_structures can use different weights for the same stochastic_scenario. Note that every stochastic_scenario that appears in the model must have a weight_relative_to_parents defined for it related to the used stochastic_structure! See the Stochastic Framework section for more information about how different stochastic_structures interact in *SpineOpt.jl*.)

Since the Stochastic Framework in *SpineOpt.jl* supports *stochastic directed acyclic graphs* instead of simple *stochastic trees*, it is possible to define stochastic_structures with converging stochastic_scenarios. In these cases, the child stochastic_scenarios inherint the weight of all of their parents, and the final weight that will appear in the Objective function is calculated as shown below:

```
# For root `stochastic_scenarios` (meaning no parents)

weight(scenario) = weight_relative_to_parents(scenario)

# If not a root `stochastic_scenario`

weight(scenario) = sum([weight(parent) * weight_relative_to_parents(scenario)] for p
```

The above calculation is performed starting from the roots, generation by generation, until the leaves of the *stochastic DAG*. Thus, the final weight of each stochastic_scenario is dependent on the weight_relative_to_parents Parameters of all its ancestors.

write_lodf_file

> A boolean flag for whether the LODF values should be written to a results file.

> **Default value:** false

> **Uses Parameter Value Lists:** boolean_value_list

> **Related Object Classes:** model

If this parameter value is set to `true`, a diagnostics file containing all the network line outage distributions factors in CSV format will be written to the current directory.

write_mps_file

> A selector for writing an .mps file of the model.

> **Uses Parameter Value Lists:** write_mps_file_list

> **Related Object Classes:** model

This parameter is deprecated and will be removed in a future version.

This parameter controls when to write a diagnostic model file in MPS format. If set to `write_mps_always`, the model will always be written in MPS format to the current directory. If set to `write\_mps\_on\_no\_solve`, the MPS file will be written when the model solve terminates with a status of false. If set to `write\_mps\_never`, no file will be written

write_ptdf_file

> A boolean flag for whether the LODF values should be written to a results file.

> **Default value:** false

> **Uses Parameter Value Lists:** boolean_value_list

> **Related Object Classes:** model

If this parameter value is set to `true`, a diagnostics file containing all the network power transfer distributions factors in CSV format will be written to the current directory.

« Relationship Classes

Parameter Value Lists »

Parameter Value Lists

balance_type_list

> **Default value:** balance_type_none

The balance_type_list parameter value list contains the possible values for the balance_type parameter.

boolean_value_list

> **Default value:** true

A list of boolean values (True or False).

commodity_physics_list

> **Default value:** commodity_physics_ptdf

_list

`commodity_physics_list` holds the possible values for the commodity parameter commodity_physics parameter. See commodity_physics for more details

connection_investment_variable_type_list

> **Default value:** connection_investment_variable_type_integer

The connection_investment_variable_type_list holds the possible values for the type of a connection's investment variable which may be chosen between integer or continuous.

connection_type_list

> **Default value:** connection_type_normal

`connection_type_list` holds the possible values for the connection_type parameter. See connection_type for more details

`constraint_sense_list`

> **Default value:** >=

The constraint_sense_list parameter value list contains the possible values for the constraint_sense parameter.

`duration_unit_list`

> **Default value:** minute

The duration_unit_list parameter value list contains the possible values for the duration_unit parameter.

`model_type_list`

> **Default value:** spineopt_other

`model_type_list` holds the possible values for the model parameter model_type parameter. See model_type for more details

`node_opf_type_list`

> **Default value:** node_opf_type_reference

Houses the different possible values for the node_opf_type parameter. To identify the reference node, set `node_opf_type` = `:node_opf_type_reference`, while `node_opf_type` = `node_opf_type_normal` is the default value for non-reference nodes.

See also powerflow.

`unit_investment_variable_type_list`

> **Default value:** unit_investment_variable_type_integer

`unit_investment_variable_type_list` holds the possible values for the type of a `unit`'s investment variable which may be chosen from integer, binary or continuous.

`unit_online_variable_type_list`

> **Default value:** unit_online_variable_type_linear

`unit_online_variable_type_list` holds the possible values for the type of a unit's commitment status variable which may be chosen from binary, integer, or linear.

`variable_type_list`

> **Default value:** variable_type_integer

The variable_type_list parameter value list contains the possible values for the connection_investment_variable_type and storage_investment_variable_type parameters.

`write_mps_file_list`

> **Default value:** write_mps_on_no_solve
> `_list`

This parameter value list is deprecated and will be removed in a future version.

Houses the different values for the write_mps_file parameter. Possible values include `write_mps_always`, `write\_mps\_on\_no\_solve`, and `write\_mps\_never`.

Variables

binary_gas_connection_flow

Math symbol: $v_{binary\_gas\_connection\_flow}$

Indices: (connection=conn, node=n, direction=d, stochastic_scenario=s, t=t)

Indices function: binary_gas_connection_flow_indices

Binary variable with the indices node n over the connection $conn$ in the direction to_node for the stochastic scenario s at timestep t describing if the direction of gas flow for a pressure drive gas transfer is in the indicated direction.

connection_flow

Math symbol: $v_{connection\_flow}$

Indices: (connection=conn, node=n, direction=d, stochastic_scenario=s, t=t)

Indices function: connection_flow_indices

Commodity flow associated with node n over the connection $conn$ in the direction d for the stochastic scenario s at timestep t

connection_intact_flow

Math symbol: $v_{connection\_intact\_flow}$

Indices: (connection=conn, node=n, direction=d, stochastic_scenario=s, t=t)

Indices function: connection_intact_flow_indices

Represents the ptdf-based flow on connections where all investment candidate connections are present in the network.

connections_decommissioned

Math symbol: $v_{connections\_decommissioned}$

Indices: (connection=conn, stochastic_scenario=s, t=t)

Indices function: connections_invested_available_indices

Number of decomissioned connections $conn$ for the stochastic scenario s at timestep t

connections_invested

Math symbol: $v_{connections\_invested}$

Indices: (connection=conn, stochastic_scenario=s, t=t)

Indices function: connections_invested_available_indices

Number of connections $conn$ invested at timestep t in for the stochastic scenario s

connections_invested_available

Math symbol: $v_{connections\_invested\_available}$

Indices: (connection=conn, stochastic_scenario=s, t=t)

Indices function: connections_invested_available_indices

Number of invested connections $conn$ that are available still the stochastic scenario s at timestep t

`mp_objective_lowerbound_indices`

Math symbol: $v_{mp\_objective\_lowerbound\_indices}$

Indices: (t=t)

Indices function: mp_objective_lowerbound_indices

Updating lowerbound for master problem of Benders decomposition

`node_injection`

Math symbol: $v_{node\_injection}$

Indices: (node=n, stochastic_scenario=s, t=t)

Indices function: node_injection_indices

Commodity injections at node n for the stochastic scenario s at timestep t

`node_pressure`

Math symbol: $v_{node\_pressure}$

> **Indices:** (node=n, stochastic_scenario=s, t=t)

> **Indices function:** node_pressure_indices

Pressue at a node n for a specific stochastic scenario s and timestep t. See also: has_pressure

node_slack_neg

> **Math symbol:** $v_{node\_slack\_neg}$

> **Indices:** (node=n, stochastic_scenario=s, t=t)

> **Indices function:** node_slack_indices

Negative slack variable at node n for the stochastic scenario s at timestep t

node_slack_pos

> **Math symbol:** $v_{node\_slack\_pos}$

> **Indices:** (node=n, stochastic_scenario=s, t=t)

> **Indices function:** node_slack_indices

Positive slack variable at node n for the stochastic scenario s at timestep t

node_state

> **Math symbol:** v_{node_state}

> **Indices:** (node=n, stochastic_scenario=s, t=t)

> **Indices function:** node_state_indices

Storage state at node n for the stochastic scenario s at timestep t

node_voltage_angle

> **Math symbol:** $v_{node\_voltage\_angle}$

> **Indices:** (node=n, stochastic_scenario=s, t=t)

> **Indices function:** node_voltage_angle_indices

Voltage angle at a node n for a specific stochastic scenario s and timestep t. See also: has_voltage_angle

nonspin_ramp_down_unit_flow

> **Math symbol:** $v_{nonspin\_ramp\_down\_unit\_flow}$

> **Indices:** (unit=u, node=n, direction=d, stochastic_scenario=s, t=t)

> **Indices function:** nonspin_ramp_down_unit_flow_indices

Non-spinning downward reserve commodity flows of unit u at node n in the direction d for the stochastic scenario s at timestep t

nonspin_ramp_up_unit_flow

> **Math symbol:** $v_{nonspin\_ramp\_up\_unit\_flow}$

> **Indices:** (unit=u, node=n, direction=d, stochastic_scenario=s, t=t)

> **Indices function:** nonspin_ramp_up_unit_flow_indices

Non-spinning upward reserve commodity flows of unit u at node n in the direction d for the stochastic scenario s at timestep t

nonspin_units_shut_down

> **Math symbol:** $v_{nonspin\_units\_shut\_down}$

> **Indices:** (unit=u, node=n, stochastic_scenario=s, t=t)

> **Indices function:** nonspin_units_shut_down_indices

Number of units u held available for non-spinning downward reserve provision via shutdown to node n for the stochastic scenario s at timestep t

nonspin_units_started_up

> **Math symbol:** $v_{nonspin\_units\_started\_up}$

> **Indices:** (unit=u, node=n, stochastic_scenario=s, t=t)

> **Indices function:** nonspin_units_started_up_indices

Number of units u held available for non-spinning upward reserve provision via startup to node n for the stochastic scenario s at timestep t

ramp_down_unit_flow

> **Math symbol:** $v_{ramp\_down\_unit\_flow}$

> **Indices:** (unit=u, node=n, direction=d, stochastic_scenario=s, t=t)

> **Indices function:** ramp_down_unit_flow_indices

Spinning downward ramp commodity flow associated with node n of unit u with node n over the connection $conn$ in the direction d for the stochastic scenario s at timestep t

`ramp_up_unit_flow`

> **Math symbol:** $v_{ramp\_up\_unit\_flow}$

> **Indices:** (unit=u, node=n, direction=d, stochastic_scenario=s, t=t)

> **Indices function:** ramp_up_unit_flow_indices

Spinning upward ramp commodity flow associated with node n of unit u with node n over the connection $conn$ in the direction d for the stochastic scenario s at timestep t

`shut_down_unit_flow`

> **Math symbol:** $v_{shut\_down\_unit\_flow}$

> **Indices:** (unit=u, node=n, direction=d, stochastic_scenario=s, t=t)

> **Indices function:** shut_down_unit_flow_indices

Downward ramp commodity flow during shutdown associated with node n of unit u with node n over the connection $conn$ in the direction d for the stochastic scenario s at timestep t

start_up_unit_flow

> **Math symbol:** $v_{start\_up\_unit\_flow}$

> **Indices:** (unit=u, node=n, direction=d, stochastic_scenario=s, t=t)

> **Indices function:** start_up_unit_flow_indices

Upward ramp commodity flow during start-up associated with node n of unit u with node n over the connection $conn$ in the direction d for the stochastic scenario s at timestep t

storages_decommissioned

> **Math symbol:** $v_{storages\_decommissioned}$

> **Indices:** (node=n, stochastic_scenario=s, t=t)

> **Indices function:** storages_invested_available_indices

Number of decomissioned storage nodes n for the stochastic scenario s at timestep t

storages_invested

> **Math symbol:** $v_{storages\_invested}$

> **Indices:** (node=n, stochastic_scenario=s, t=t)

> **Indices function:** storages_invested_available_indices

Number of storage nodes n invested in at timestep t for the stochastic scenario s

storages_invested_available

> **Math symbol:** $v_{storages\_invested\_available}$

> **Indices:** (node=n, stochastic_scenario=s, t=t)

> **Indices function:** storages_invested_available_indices

Number of invested storage nodes n that are available still the stochastic scenario s at timestep t

unit_flow

> **Math symbol:** v_{unit_flow}

> **Indices:** (unit=u, node=n, direction=d, stochastic_scenario=s, t=t)

> **Indices function:** unit_flow_indices

Commodity flow associated with node n over the unit u in the direction d for the stochastic scenario s at timestep t

unit_flow_op

> **Math symbol:** $v_{unit\_flow\_op}$

> **Indices:** (unit=u, node=n, direction=d, i=i, stochastic_scenario=s, t=t)

> **Indices function:** unit_flow_op_indices

Contribution of the unit flow assocaited with operating point i

units_available

> **Math symbol:** $v_{units\_available}$

> **Indices:** (unit=u, stochastic_scenario=s, t=t)

> **Indices function:** units_on_indices

Number of available units u for the stochastic scenario s at timestep t

units_invested

> **Math symbol:** $v_{units\_invested}$

> **Indices:** (unit=u, stochastic_scenario=s, t=t)

> **Indices function:** units_invested_available_indices

Number of units u for the stochastic scenario s invested in at timestep t

units_invested_available

> **Math symbol:** $v_{units\_invested\_available}$

> **Indices:** (unit=u, stochastic_scenario=s, t=t)

> **Indices function:** units_invested_available_indices

Number of invested units u that are available still the stochastic scenario s at timestep t

units_mothballed

> **Math symbol:** $v_{units\_mothballed}$

> **Indices:** (unit=u, stochastic_scenario=s, t=t)

> **Indices function:** units_invested_available_indices

Number of units u for the stochastic scenariocenario s mothballed at timestep t

units_on

> **Math symbol:** v_{units_on}

> **Indices:** (unit=u, stochastic_scenario=s, t=t)

> **Indices function:** units_on_indices

Number of online units u for the stochastic scenario s at timestep t

units_shut_down

> **Math symbol:** $v_{units\_shut\_down}$

> **Indices:** (unit=u, stochastic_scenario=s, t=t)

> **Indices function:** units_on_indices

Number of units u for the stochastic scenario s that switched to offline status at timestep t

units_started_up

> **Math symbol:** $v_{units\_started\_up}$

> **Indices:** (unit=u, stochastic_scenario=s, t=t)

> **Indices function:** units_on_indices

Number of units u for the stochastic scenario s that switched to online status at timestep t

---

« Parameter Value Lists

Constraints »

Constraints

Balance constraint

Nodal balance

In **SpineOpt**, node is the place where an energy balance is enforced. As universal aggregators, they are the glue that brings all components of the energy system together. An energy balance is created for each node for all `node_stochastic_time_indices`, unless the balance_type parameter of the node takes the value balance_type_none or if the node in question is a member of a node group, for which the balance_type is balance_type_group. The parameter nodal_balance_sense defaults to equality, but can be changed to allow overproduction (nodal_balance_sense `>=`) or underproduction (nodal_balance_sense `<=`). The energy balance is enforced by the following constraint:

$$
\begin{aligned}
& v_{node\_injection}(n, s, t) \\
& + \sum_{\substack{(conn,n',d_{in},s,t) \in connection\_flow\_indices: \\ d_{out} == :to\_node}} v_{connection\_flow}(conn, n', d_{in}, s, t) \\
& - \sum_{\substack{(conn,n',d_{out},s,t) \in connection\_flow\_indices: \\ d_{out} == :from\_node}} v_{connection\_flow}(conn, n', d_{out}, s, t) \\
& + v_{node\_slack\_pos}(n, s, t) \\
& - v_{node\_slack\_neg}(n, s, t) \\
& \{>=, ==, <=\} \\
& 0 \\
& \forall (n, s, t) \in node\_stochastic\_time\_indices : \\
& p_{balance\_type}(n) ! = balance\_type\_none \\
& \nexists ng \in groups(n) : balance\_type\_group
\end{aligned}
$$

The constraint consists of the node injections, the net connection_flows and node slack variables.

Node injection

The node injection itself represents all local production and consumption, represented by the sum of all connected unit flows and the nodal demand. The node injection is created for each node in the network (unless the node is only used for parameter aggregation purposes, see Introduction to groups of objects).

$$
\begin{aligned}
& v_{node\_injection}(n, s, t) \\
& == \\
& + \sum_{\substack{(u,n',d_{in},s,t) \in unit\_flow\_indices: \\ d_{out} == :to\_node}} v_{unit\_flow}(u, n', d_{in}, s, t) \\
& - \sum_{\substack{(u,n',d_{out},s,t) \in unit\_flow\_indices: \\ d_{out} == :from\_node}} v_{unit\_flow}(u, n', d_{out}, s, t) \\
& - p_{demand}(n, s, t) \\
& \forall (n, s, t) \in node\_stochastic\_time\_indices
\end{aligned}
$$

Node injection with storage capability

If a node corresponds to a storage node, the parameter has_state should be set to true for this node. In this case the nodal injection will translate to the following constraint:

$$
\begin{aligned}
& v_{node\_injection}(n,s,t) \\
& == \\
& (v_{node\_state}(n,s,t\_before) \\
& - v_{node\_state}(n,s,t) \cdot p_{state\_coeff}(n,s,t)) \\
& / \Delta t_{after} \\
& - v_{node\_state}(n,s,t) \cdot p_{frac\_state\_loss}(n,s,t) \\
& + \sum_{\substack{(n2,s,t) \in node\_state\_indices: \\ \exists diff\_coeff(n2,n)}} v_{node\_state}(n2,s,t) \\
& - \sum_{\substack{(n2,s,t) \in node\_state\_indices: \\ \exists diff\_coeff(n,n2)}} v_{node\_state}(n2,s,t) \\
& + \sum_{\substack{(u,n',d_{in},s,t) \in unit\_flow\_indices: \\ d_{out} == :to\_node}} v_{unit\_flow}(u,n',d_{in},s,t) \\
& - \sum_{\substack{(u,n',d_{out},s,t) \in unit\_flow\_indices: \\ d_{out} == :from\_node}} v_{unit\_flow}(u,n',d_{out},s,t) \\
& - demand(n,s,t) \\
& \forall (n,t) \in node\_time\_indices : p_{has\_state}(n) \\
& \forall s \in stochastic\_scenario\_path \\
& t_{before} \in t\_before\_t(t\_after = t)
\end{aligned}
$$

Note that for simplicity, the stochastic path is assumed to be known. In the constraint `constraint_node_injection.jl` the active stochastic paths of all involved variables is retrieved beforehand.

Node state capacity

To limit the storage content, the v_{node_state} variable needs be constrained by the following equation:

$$
\begin{aligned}
& v_{node\_state}(n,s,t) \\
& <= p_{node\_state\_cap}(n,s,t) \\
& \forall (n,s,t) \in node\_stochastic\_time\_indices : \\
& p_{has\_state}(n)
\end{aligned}
$$

The discharging and charging behavior of storage nodes can be described through unit(s), representing the link between the storage node and the supply node. Note that the dis-/charging efficiencies and capacities are properties of these units. See the capacity constraint and the unit flow ratio constraints

Cyclic condition on node state variable

To ensure that the node state at the end of the optimization is at least the same value as the initial value at the beginning of the optimization (or higher), the cyclic node state constraint can be used by setting the cyclic_condition of a node__temporal_block to `true`. This triggers the following cyclic constraint:

$$
\begin{aligned}
&v_{node\_state}(n, s, t) \\
&>= v_{node\_state}(n, s, t) \\
&\forall (n, tb) \in p_{cyclic\_condition}(n, tb) : \\
&\{p_{cyclic\_condition}(n, tb) == true, \\
&p_{has\_state}(n)\} \\
&\forall (n', t_{initial}) \in node\_time\_indices : \\
&\{n' == n, \\
&t_{initial} == t\_before\_t(t\_after = first(t \in tb)), \\
&\forall (n', t_{last}) \in node\_time\_indices : \\
&n' == n, \\
&t_{last} == last(t \in tb)) \\
&\forall s \in stochastic\_path
\end{aligned}
$$

Unit operation

In the following, the operational constraints on the variables associated with units will be elaborated on. The static constraints, in contrast to the dynamic constraints, are addressing constraints without sequential time-coupling. It should however be noted that static constraints can still perform temporal aggregation.

Static constraints

The fundamental static constraints for units within SpineOpt relate to the relationships between commodity flows from and to units and to limits on the unit flow capacity.

Conversion constraint / limiting flow shares inprocess / relationship in process

A unit can have different commodity flows associated with it. The most simple relationship between these flows is a linear relationship between input and/or output nodes/node groups. SpineOpt holds constraints for each combination of flows and also for the type of relationship, i.e. whether it is a maximum, minimum or fixed ratio between commodity flows. Note that node groups can be used in order to aggregate flows, i.e. to give a ratio between a combination of units flows.

Ratios between output and input flows of a unit

By defining the parameters fix_ratio_out_in_unit_flow, max_ratio_out_in_unit_flow or min_ratio_out_in_unit_flow, a ratio can be set between **out**going and **in**coming flows from and to a unit. Whenever there is only a single input node and a single output node, this relationship relates to the notion of an efficiency. Also, the ratio equation can for instance be used to relate emissions to input primary fuel flows. In the most general form of the equation, two node groups are defined (an input node group ng_{in} and an output node group ng_{out}), and a linear relationship is expressed between both node groups. Note that whenever the relationship is specified between groups of multiple nodes, there remains a degree of freedom regarding the composition of the input node flows within group ng_{in} and the output node flows within group ng_{out}.

The constraint given below enforces a fixed, maximum or minimum ratio between outgoing and incoming unit_flow. Note that the potential node groups, that the parameters fix_ratio_out_in_unit_flow, max_ratio_out_in_unit_flow and min_ratio_out_in_unit_flow defined on, are getting internally expanded to the members of the node group within the unit_flow_indices.

$$
\begin{aligned}
&\sum_{\substack{(u,n,d,s,t_{out}) \in unit\_flow\_indices: \\ (u,n,d,s,t_{out}) \in (u,ng_{out},:to\_node,s,t)}} v_{unit\_flow}(u,n,d,s,t_{out}) \cdot \Delta t_{out} \\
&\{ \\
&== p_{fix\_ratio\_out\_in\_unit\_flow}(u, ng_{out}, ng_{in}, s, t), \\
&<= p_{max\_ratio\_out\_in\_unit\_flow}(u, ng_{out}, ng_{in}, s, t), \\
&>= p_{min\_ratio\_out\_in\_unit\_flow}(u, ng_{out}, ng_{in}, s, t) \\
&\} \\
&\cdot \sum_{\substack{(u,n,d,s,t_{in}) \in unit\_flow\_indices: \\ (u,n,d,s,t_{in}) \in (u,ng_{in},:from\_node,s,t)}} v_{unit\_flow}(u,n,d,s,t_{in}) \cdot \Delta t_{in} \\
&+ p_{\{fix,max,min\}\_units\_on\_coefficient\_out\_in}(u, ng_{out}, ng_{in}, s, t) \\
&\sum_{\substack{(u,s,t_{units\_on}) \in units\_on\_indices: \\ (u,s,t_{units\_on}) \in (u,s,t)}} v_{units\_on}(u,s,t_{units\_on}) \\
&\cdot \min(\Delta t_{units\_on}, \Delta t) \\
&\forall (u, ng_{out}, ng_{in}) \in ind(p_{\{fix,max,min\}\_ratio\_out\_in\_unit\_flow}), \\
&\forall t \in time\_slices, \forall s \in stochastic\_path
\end{aligned}
$$

Note that a right-hand side constant coefficient associated with the variable units_on can optionally be included, triggered by the existence of the fix_units_on_coefficient_out_in, max_units_on_coefficient_out_in, min_units_on_coefficient_out_in, respectively.

Ratios between input and output flows of a unit

Similarly to the ratio between outgoing and incoming unit flows, a ratio can also be defined in reverse between **in**coming and **out**going flows.

$$
\begin{aligned}
&\sum_{\substack{(u,n,d,s,t_{in}) \in unit\_flow\_indices: \\ (u,n,d,s,t_{in}) \in (u,ng_{in},:from\_node,s,t)}} v_{unit\_flow}(u,n,d,s,t_{in}) \cdot \Delta t_{in} \\
&\{ \\
&== p_{fix\_ratio\_in\_out\_unit\_flow}(u, ng_{in}, ng_{out}, s, t), \\
&<= p_{max\_ratio\_in\_out\_unit\_flow}(u, ng_{in}, ng_{out}, s, t), \\
&>= p_{min\_ratio\_in\_out\_unit\_flow}(u, ng_{in}, ng_{out}, s, t) \\
&\} \\
&\cdot \sum_{\substack{(u,n,d,s,t_{out}) \in unit\_flow\_indices: \\ (u,n,d,s,t_{in}) \in (u,ng_{in},:to\_node,s,t)}} v_{unit\_flow}(u,n,d,s,t_{out}) \cdot \Delta t_{out} \\
&+ p_{\{fix,max,min\}\_units\_on\_coefficient\_in\_out}(u, ng_{in}, ng_{out}, s, t) \\
&\sum_{\substack{(u,s,t_{units\_on}) \in units\_on\_indices: \\ (u,s,t_{units\_on}) \in (u,s,t)}} v_{units\_on}(u,s,t_{units\_on}) \\
&\cdot \min(\Delta t_{units\_on}, \Delta t) \\
&\forall (u, ng_{in}, ng_{out}) \in ind(p_{\{fix,max,min\}\_ratio\_in\_out\_unit\_flow}), \\
&\forall t \in time\_slices, \forall s \in stochastic\_path
\end{aligned}
$$

Note that a right-hand side constant coefficient associated with the variable units_on can optionally be included, triggered by the existence of the fix_units_on_coefficient_in_out, max_units_on_coefficient_in_out, min_units_on_coefficient_in_out, respectively.

Ratios between input and input flows of a unit

Similarly to the ratio between outgoing and incoming units flows, one can also define a fixed, maximum or minimum ratio between **in**coming flows of a units.

$$
\begin{aligned}
&\sum_{\substack{(u,n,d,s,t_{in1}) \in unit\_flow\_indices: \\ (u,n,d,s,t_{in1}) \in (u,ng_{in1},:from\_node,s,t)}} v_{unit\_flow}(u,n,d,s,t_{in1}) \cdot \Delta t_{in1} \\
&\{ \\
&== p_{fix\_ratio\_in\_in\_unit\_flow}(u, ng_{in1}, ng_{in2}, s, t), \\
&<= p_{max\_ratio\_in\_in\_unit\_flow}(u, ng_{in1}, ng_{in2}, s, t), \\
&>= p_{min\_ratio\_in\_in\_unit\_flow}(u, ng_{in1}, ng_{in2}, s, t) \\
&\} \\
&\cdot \sum_{\substack{(u,n,d,s,t_{in2}) \in unit\_flow\_indices: \\ (u,n,d,s,t_{in2}) \in (u,ng_{in2},:from\_node,s,t)}} v_{unit\_flow}(u,n,d,s,t_{in2}) \cdot \Delta t_{in2} \\
&+ p_{\{fix,max,min\}\_units\_on\_coefficient\_in\_in}(u, ng_{in1}, ng_{in2}, s, t) \\
&\sum_{\substack{(u,s,t_{units\_on}) \in units\_on\_indices: \\ (u,s,t_{units\_on}) \in (u,s,t)}} v_{units\_on}(u,s,t_{units\_on}) \\
&\cdot \min(\Delta t_{units\_on}, \Delta t) \\
&\forall (u, ng_{in1}, ng_{in2}) \in ind(p_{\{fix,max,min\}\_ratio\_in\_in\_unit\_flow}), \\
&\forall t \in time\_slices, \forall s \in stochastic\_path
\end{aligned}
$$

Note that a right-hand side constant coefficient associated with the variable units_on can optionally be included, triggered by the existence of the fix_units_on_coefficient_in_in, max_units_on_coefficient_in_in, min_units_on_coefficient_in_in, respectively.

Ratios between output and output flows of a unit

Similarly to the ratio between outgoing and incoming units flows, one can also define a fixed, maximum or minimum ratio between **out**going flows of a units.

$$
\begin{aligned}
&\sum_{\substack{(u,n,d,s,t_{out1}) \in unit\_flow\_indices: \\ (u,n,d,s,t_{out1}) \in (u,ng_{out1},:to\_node,s,t)}} v_{unit\_flow}(u,n,d,s,t_{out1}) \cdot \Delta t_{out1} \\
&\{ \\
&== p_{fix\_ratio\_out\_out\_unit\_flow}(u, ng_{out1}, ng_{out2}, s, t), \\
&<= p_{max\_ratio\_out\_out\_unit\_flow}(u, ng_{out1}, ng_{out2}, s, t), \\
&>= p_{min\_ratio\_out\_out\_unit\_flow}(u, ng_{out1}, ng_{out2}, s, t) \\
&\} \\
&\cdot \sum_{\substack{(u,n,d,s,t_{out2}) \in unit\_flow\_indices: \\ (u,n,d,s,t_{out2}) \in (u,ng_{out2},:to\_node,s,t)}} v_{unit\_flow}(u,n,d,s,t_{out2}) \cdot \Delta t_{out2} \\
&+ p_{\{fix,max,min\}\_units\_on\_coefficient\_out\_out}(u, ng_{out1}, ng_{out2}, s, t) \\
&\sum_{\substack{(u,s,t_{units\_on}) \in units\_on\_indices: \\ (u,s,t_{units\_on}) \in (u,s,t)}} v_{units\_on}(u,s,t_{units\_on}) \\
&\cdot \min(\Delta t_{units\_on}, \Delta t) \\
&\forall (u, ng_{out1}, ng_{out2}) \in ind(p_{\{fix,max,min\}\_ratio\_out\_out\_unit\_flow}), \\
&\forall t \in time\_slices, \forall s \in stochastic\_path
\end{aligned}
$$

Note that a right-hand side constant coefficient associated with the variable units_on can optionally be included, triggered by the existence of the fix_units_on_coefficient_out_out, max_units_on_coefficient_out_out, min_units_on_coefficient_out_out, respectively.

Bounds on the unit capacity

In a multi-commodity setting, there can be different commodities entering/leaving a certain technology/unit. These can be energy-related commodities (e.g., electricity, natural gas, etc.), emissions, or other commodities (e.g., water, steel). The unit_capacity be specified for at least one unit__to_node or unit__from_node relationship, in order to trigger a constraint on the maximum commodity flows to this location in each time step. When desirable, the capacity can be specified for a group of nodes (e.g. combined capacity for multiple products).

$$
\begin{aligned}
&\sum_{\substack{(u,n,d,s,t') \in unit\_flow\_indices: \\ (u,n,d,s,t') \in (u,ng,d,s,t)}} v_{unit\_flow}(u,n,d,s,t') \cdot \Delta t' \\
&<= p_{unit\_capacity}(u,ng,d,s,t) \\
&\cdot p_{unit\_conv\_cap\_to\_flow}(u,ng,d,s,t) \\
&\cdot \sum_{\substack{(u,s,t_{units\_on}) \in units\_on\_indices: \\ (u,\Delta t_{units\_on} \in (u,t)}} v_{units\_on}(u,s,t_{units\_on}) \\
&\cdot \min(t_{units\_on}, \Delta t) \\
&\forall (u,ng,d) \in ind(p_{unit\_capacity}), \\
&\forall t \in time\_slices, \\
&\forall s \in stochastic\_path
\end{aligned}
$$

Note that the conversion factor unit_conv_cap_to_flow has a default value of `1`, but can be adjusted in case the unit of measurement for the capacity is different to the unit flows unit of measurement.

Dynamic constraints

Commitment constraints

For modeling certain technologies/units, it is important to not only have unit_flow variables of different commodities, but also model the online ("commitment") status of the unit/technology at every time step. Therefore, an additional variable units_on is introduced. This variable represents the number of online units of that technology (for a normal unit commitment model, this variable might be a binary, for investment planning purposes, this might also be an integer or even a continuous variable). To define the type of a commitment variable, see online_variable_type. Commitment variables will be introduced by the following constraints (with corresponding parameters):

- constraint on `units_on`
- constraint on `units_available`
- constraint on the unit state transition
- constraint on the minimum operating point
- constraint on minimum down time
- constraint on minimum up time
- constraint on ramp rates

Bound on online units

The number of online units need to be restricted to the number of available units:

$$
\begin{aligned}
&v_{units\_on}(u,s,t) \\
&<= v_{units\_available}(u,s,t) \\
&\forall (u,s,t) \in units\_on\_indices
\end{aligned}
$$

Bound on available units

The number of available units itself is constrained by the parameters unit_availability_factor and number_of_units, and the variable number of invested units units_invested_available:

$$
\begin{aligned}
&v_{units\_available}(u,s,t) \\
&== p_{unit\_availability\_factor}(u,s,t) \\
&\cdot (p_{number\_of\_units}(u,s,t) \\
&+ \sum_{(u,s,t) \in units\_invested\_available\_indices} v_{units\_invested\_available}(u,s,t)) \\
&\forall (u,s,t) \in units\_on\_indices
\end{aligned}
$$

The investment formulation is described in chapter Investments.

Unit state transition

The units on status is constrained by shutting down and starting up actions. This transition is defined as follows:

$$
\begin{aligned}
&v_{units\_on}(u, s, t_{after}) \\
&- v_{units\_started\_up}(u, s, t_{after}) \\
&+ v_{units\_shut\_down}(u, s, t_{after}) \\
&== v_{units\_on}(u, s, t_{before}) \\
&\forall (u, s, t_{after}) \in units\_on\_indices, \\
&\forall t_{before} \in t\_before\_t(t\_after = t_{after}) : t_{before} \in units\_on\_indices
\end{aligned}
$$

Constraint on minimum operating point

The minimum operating point of a unit can be based on the unit_flows of input or output nodes/node groups ng:

$$
\begin{aligned}
&\sum_{\substack{(u,n,d,s,t') \in unit\_flow\_indices: \\ (u,n,d,t') \in (u,ng,d,t)}} v_{unit\_flow}(u, n, d, s, t') \cdot \Delta t' \\
&>= p_{minimum\_operating\_point}(u, ng, d, s, t) \\
&\cdot p_{unit\_capacity}(u, ng, d, s, t) \\
&\cdot p_{conv\_cap\_to\_flow}(u, ng, d, s, t) \\
&\cdot \sum_{\substack{(u,s,t_{units\_on}) \in units\_on\_indices: \\ (u,\Delta t_{units\_on} \in (u,t)}} v_{units\_on}(u, s, t_{units\_on}) \\
&\cdot \min(\Delta t_{units\_on}, \Delta t) \\
&\forall (u, ng, d) \in ind(p_{minimum\_operating\_point}), \\
&\forall t \in t\_lowest\_resolution(node\_\_temporal\_block(node = members(ng))), \\
&\forall s \in stochastic\_path
\end{aligned}
$$

Note that this constraint is always generated for the lowest resolution of all involved members of the node group `ng`, i.e. the lowest resolution of the involved units flows. This is also why the term $\min(\Delta t_{units\_on}, \Delta t)$ is added for the units on variable, in order to dis-/aggregate the units on resolution to the resolution of the unit flows.

Minimum down time (basic version)

In order to impose a minimum offline time of a unit, before it can be started up again, the min_down_time parameter needs to be defined, which triggers the generation of the following constraint:

$$
\begin{aligned}
&v_{units\_available}(u, s, t) \\
&- v_{units\_on}(u, s, t) \\
&>= \sum_{\substack{(u,s,t') \in units\_on\_indices: \\ t' >= t - p_{min\_down\_time}(u,s,t) \quad t' <= t}} v_{units\_shut\_down}(u, s, t') \\
&\forall (u, s, t) \in units\_on\_indices
\end{aligned}
$$

Note that for the use reserves the generated minimum down time constraint will include startups for non-spinning reserves.

Minimum up time (basic version)

Similarly to the minimum down time constraint, a minimum time that a unit needs to remain online after a startup can be imposed by defining the min_up_time parameter. This will trigger the generation of the following constraint:

$$
\begin{aligned}
&v_{units\_on}(u, s, t) \\
&>= \sum_{\substack{(u,s,t') \in units\_on\_indices: \\ t' >= t - p_{min\_up\_time}(u,s,t), \\ t' <= t}} v_{units\_started\_up}(u, s, t') \\
&\forall (u, s, t) \in units\_on\_indices
\end{aligned}
$$

This constraint can be extended to the use of nonspinning reserves. See also.

Ramping and reserve constraints

To include ramping and reserve constraints, it is a pre requisite that minimum operating points and maximum capacity constraints are enforced as described.

For dispatchable units, additional ramping constraints can be introduced. For setting up ramping characteristics of units see Ramping and Reserves. First, the unit flows are split into their online, start-up, shut-down and non-spinning ramping contributions.

Splitting unit flows into ramps

$$
\begin{aligned}
&+ \sum_{\substack{(u,n,d,s,t_{after}) \in unit\_flow\_indices: \\ (u,n,d,t_{after}) \in (u,n,d,t_{after}) \\ !p_{is\_reserve}(n)}} v_{unit\_flow}(u,n,d,s,t_{after}) \\
&+ \sum_{\substack{(u,n,d,s,t_{after}) \in unit\_flow\_indices: \\ (u,n,d,t_{after}) \in (u,n,d,t_{after}) \\ p_{is\_reserve(n)} \\ p_{upward\_reserve}(n)}} v_{unit\_flow}(u,n,d,s,t_{after}) \\
&- \sum_{\substack{(u,n,d,s,t_{after}) \in unit\_flow\_indices: \\ (u,n,d,t_{after}) \in (u,n,d,t_{after}) \\ p_{is\_reserve(n)} \\ p_{downward\_reserve}(n)}} v_{unit\_flow}(u,n,d,s,t_{after}) \\
&- \sum_{\substack{(u,n,d,s,t_{before}) \in unit\_flow\_indices: \\ (u,n,d,t_{before}) \in (u,n,d,t_{before}) \\ !p_{is\_reserve}(n)}} v_{unit\_flow}(u,n,d,s,t_{before}) \\
&== \\
&+ \sum_{\substack{(u,n,d,s,t_{after}) \in ramp\_up\_unit\_flow\_indices: \\ (u,n,d,t_{after}) \in (u,n,d,t_{after})}} v_{ramp\_up\_unit\_flow}(u,n,d,s,t_{after}) \\
&+ \sum_{\substack{(u,n,d,s,t_{after}) \in start\_up\_unit\_flow\_indices: \\ (u,n,d,t_{after}) \in (u,n,d,t_{after})}} v_{start\_up\_unit\_flow}(u,n,d,s,t_{after}) \\
&+ \sum_{\substack{(u,n,d,s,t_{after}) \in nonspin\_ramp\_up\_unit\_flow\_indices: \\ (u,n,d,t_{after}) \in (u,n,d,t_{after})}} v_{nonspin\_ramp\_up\_unit\_flow}(u,n,d,s,t_{after}) \\
&- \sum_{\substack{(u,n,d,s,t_{after}) \in ramp\_down\_unit\_flow\_indices: \\ (u,n,d,t_{after}) \in (u,n,d,t_{after})}} v_{ramp\_down\_unit\_flow}(u,n,d,s,t_{after}) \\
&- \sum_{\substack{(u,n,d,s,t_{after}) \in shut\_down\_unit\_flow\_indices: \\ (u,n,d,t_{after}) \in (u,n,d,t_{after})}} v_{shut\_down\_unit\_flow}(u,n,d,s,t_{after}) \\
&- \sum_{\substack{(u,n,d,s,t_{after}) \in nonspin\_ramp\_down\_unit\_flow\_indices: \\ (u,n,d,t_{after}) \in (u,n,d,t_{after})}} v_{nonspin\_ramp\_down\_unit\_flow}(u,n,d,s,t_{after}) \\
&\forall (u,n,d,s,t_{after}) \in (\\
&ramp\_up\_unit\_flow\_indices, \\
&start\_up\_unit\_flow\_indices, \\
&nonspin\_ramp\_up\_unit\_flow\_indices, \\
&ramp\_down\_unit\_flow\_indices, \\
&shut\_down\_unit\_flow\_indices, \\
&nonspin\_ramp\_down\_unit\_flow\_indices) \\
&\forall t_{before} \in t\_before\_t(t\_after = t_{after}) : t_{before} \in unit\_flow\_indices
\end{aligned}
$$

Note that each *individual* tuple of the `unit_flow_indices` is split into its ramping contributions, if any of the ramping variables exist for this tuple. How to set-up ramps for units is described in Ramping and Reserves.

Constraint on spinning upwards ramp_up

The maximum online ramp up ability of a unit can be constraint by the ramp_up_limit, expressed as a share of the unit_capacity. With this constraint, online (i.e. spinning) ramps can be applied to groups of commodities (e.g. electricity + balancing capacity). Moreover, balancing product might have specific ramping requirements, which can herewith also be enforced.

$$
\begin{aligned}
&+ \sum_{\substack{(u,n,d,s,t) \in ramp\_up\_unit\_flow\_indices: \\ (u,n,d) \in (u,ng,d)}} v_{ramp\_up\_unit\_flow}(u,n,d,s,t) \\
&<= \\
&+ \sum_{\substack{(u,s,t') \in units\_on\_indices: \\ (u,s) \in (u,s) \\ t' \in t\_overlap\_t(t)}} (v_{units\_on}(u,s,t') - v_{units\_started\_up}(u,s,t')) \\
&\min(\Delta t', \Delta t) \\
&\cdot p_{ramp\_up\_limit}(u,ng,d,s,t) \\
&\cdot p_{unit\_capacity}(u,ng,d,s,t) \\
&\cdot p_{conv\_cap\_to\_flow}(u,ng,d,s,t) \\
&\forall (u,ng,d) \in ind(p_{ramp\_up\_limit}) \\
&\forall s \in stochastic\_path, \forall t \in time\_slice
\end{aligned}
$$

Note that only online units that are not started up during this timestep are considered.

Constraint on minimum upward start up ramp_up

To enforce a lower bound on the ramp of a unit during start-up, the min_startup_ramp given as a share of the unit_capacity needs to be defined, which triggers the constraint below. Usually, only non-reserve commodities can have a start-up ramp. However, it is possible to include them, by adding them to the ramp defining node `ng`.

$$
\begin{aligned}
&+ \sum_{\substack{(u,n,d,s,t) \in start\_up\_unit\_flow\_indices: \\ (u,n,d) \in (u,ng,d)}} v_{start\_up\_unit\_flow}(u,n,d,s,t) \\
&>= \\
&+ \sum_{\substack{(u,s,t') \in units\_on\_indices: \\ (u,s) \in (u,s) \\ t' \in t\_overlap\_t(t)}} v_{units_started\_up}(u,s,t) \\
&\cdot p_{min\_startup\_ramp}(u,ng,d,s,t) \\
&\cdot p_{unit\_capacity}(u,ng,d,s,t) \\
&\cdot p_{conv\_cap\_to\_flow}(u,ng,d,s,t) \\
&\forall (u,ng,d) \in ind(p_{min\_startup\_ramp}) \\
&\forall s \in stochastic\_path, \forall t \in time\_slice
\end{aligned}
$$

Constraint on maximum upward start up ramp_up

This constraint enforces a upper limit on the unit ramp during startup process, triggered by the existence of the max_startup_ramp, which should be given as a share of the unit_capacity. Typically, only ramp flows to non-reserve nodes are considered during the start-up process. However, it is possible to include them, by adding them to the ramp defining node `ng`.

$$
\begin{aligned}
&+ \sum_{\substack{(u,n,d,s,t) \in start\_up\_unit\_flow\_indices: \\ (u,n,d) \in (u,ng,d)}} v_{start\_up\_unit\_flow}(u,n,d,s,t) \\
&<= \\
&+ \sum_{\substack{(u,s,t') \in units\_on\_indices: \\ (u,s) \in (u,s) \\ t' \in t\_overlap\_t(t)}} v_{units_started\_up}(u,s,t) \\
&\cdot p_{max\_startup\_ramp}(u,ng,d,s,t) \\
&\cdot p_{unit\_capacity}(u,ng,d,s,t) \\
&\cdot p_{conv\_cap\_to\_flow}(u,ng,d,s,t) \\
&\forall (u,ng,d) \in ind(p_{max\_startup\_ramp}) \\
&\forall s \in stochastic\_path, \forall t \in time\_slice
\end{aligned}
$$

Constraint on upward non-spinning start ups

For non-spinning reserve provision, offline units can be scheduled to provide nonspinning reserves, if they have recovered their minimum down time. If nonspinning reserves are used for a unit, the minimum down-time constraint takes the following form:

$$
\begin{aligned}
& v_{units\_available}(u,s,t) \\
& - v_{units\_on}(u,s,t) \\
& >= \sum_{\substack{(u,s,t') \in units\_on\_indices: \\ t' > t - p_{min\_down\_time}(u,s,t) \\ t' <= t}} v_{units\_shut\_down}(u,s,t') \\
& + \sum_{\substack{(u',n',s',t') \in nonspin\_units\_started\_up\_indices: \\ (u',s',t') \in (u,s,t)}} v_{nonspin\_units\_started\_up}(u',n',s',t') \\
& \forall (u,s,t) \in units\_on\_indices : \\
& (u,n,s,t) \in nonspin\_units\_started\_up\_indices
\end{aligned}
$$

Minimum nonspinning ramp up

The nonspinning ramp flows of a units nonspin_ramp_up_unit_flow are dependent on the units holding available for nonspinning reserve provision, i.e. nonspin_units_started_up. A lower bound on these nonspinning reserves can be enforced by defining the min_res_startup_ramp parameter (given as a fraction of the unit_capacity).

$$
\begin{aligned}
& + \sum_{\substack{(u,n,d,s,t) \in nonspin\_ramp\_up\_unit\_flow\_indices: \\ (u,n,d) \in (u,ng,d)}} v_{nonspin\_ramp\_up\_unit\_flow}(u,n,d,s,t) \\
& >= \\
& + \sum_{\substack{(u,n,s,t) \in nonspin\_units\_started\_up\_indices: \\ (u,n) \in (u,ng}} v_{nonspin\_units\_started\_up}(u,n,s,t) \\
& \cdot p_{min\_res\_startup\_ramp}(u,ng,d,s,t) \\
& \cdot p_{unit\_capacity}(u,ng,d,s,t) \\
& \cdot p_{conv\_cap\_to\_flow}(u,ng,d,s,t) \\
& \forall (u,ng,d) \in ind(p_{min\_res\_startup\_ramp}) \\
& \forall s \in stochastic\_path, \forall t \in time\_slice
\end{aligned}
$$

Maximum nonspinning ramp up

The nonspinning ramp flows of a units nonspin_ramp_up_unit_flow are dependent on the units holding available for nonspinning reserve provision, i.e. nonspin_units_started_up. An upper bound on these nonspinning reserves can be enforced by defining the max_res_startup_ramp parameter (given as a fraction of the unit_capacity).

$$
\begin{aligned}
& + \sum_{\substack{(u,n,d,s,t) \in nonspin\_ramp\_up\_unit\_flow\_indices: \\ (u,n,d) \in (u,ng,d)}} v_{nonspin\_ramp\_up\_unit\_flow}(u,n,d,s,t) \\
& <= \\
& + \sum_{\substack{(u,n,s,t) \in nonspin\_units\_started\_up\_indices: \\ (u,n) \in (u,ng}} v_{nonspin\_units\_started\_up}(u,n,s,t) \\
& \cdot p_{max\_res\_startup\_ramp}(u,ng,d,s,t) \\
& \cdot p_{unit\_capacity}(u,ng,d,s,t) \\
& \cdot p_{conv\_cap\_to\_flow}(u,ng,d,s,t) \\
& \forall (u,ng,d) \in ind(p_{max\_res\_startup\_ramp}) \\
& \forall s \in stochastic\_path, \forall t \in time\_slice
\end{aligned}
$$

Constraint on spinning downward ramps

Similarly to the online ramp up capbility of a unit, it is also possible to impose an upper bound on the online ramp down ability of unit by defining a ramp_down_limit, expressed as a share of the unit_capacity.

$$
\begin{aligned}
&+ \sum_{\substack{(u,n,d,s,t) \in ramp\_down\_unit\_flow\_indices: \\ (u,n,d) \in (u,ng,d)}} v_{ramp\_down\_unit\_flow}(u,n,d,s,t) \\
&<= \\
&+ \sum_{\substack{(u,s,t') \in units\_on\_indices: \\ (u,s) \in (u,s) \\ t' \in t\_overlap\_t(t)}} (v_{units\_on}(u,s,t') - v_{units\_started\_up}(u,s,t')) \\
&\cdot p_{ramp\_down\_limit}(u,ng,d,s,t) \\
&\cdot p_{unit\_capacity}(u,ng,d,s,t) \\
&\cdot p_{conv\_cap\_to\_flow}(u,ng,d,s,t) \\
&\forall (u,ng,d) \in ind(p_{ramp\_down\_limit}) \\
&\forall s \in stochastic\_path, \forall t \in time\_slice
\end{aligned}
$$

Lower bound on downward shut-down ramps

This constraint enforces a lower bound on the unit ramp during shutdown process. Usually, units will only provide shutdown ramps to non-reserve nodes. However, it is possible to include them, by adding them to the ramp defining node `ng`. The constraint is triggered by the existence of the min_shutdown_ramp parameter.

$$
\begin{aligned}
&+ \sum_{\substack{(u,n,d,s,t) \in shut\_down\_unit\_flow\_indices: \\ (u,n,d) \in (u,ng,d)}} v_{shut\_down\_unit\_flow}(u,n,d,s,t) \\
&<= \\
&+ \sum_{\substack{(u,s,t') \in units\_on\_indices: \\ (u,s) \in (u,s) \\ t' \in t\_overlap\_t(t)}} v_{units\_shut\_down}(u,s,t') \\
&\cdot p_{min\_shutdown\_ramp}(u,ng,d,s,t) \\
&\cdot p_{unit\_capacity}(u,ng,d,s,t) \\
&\cdot p_{conv\_cap\_to\_flow}(u,ng,d,s,t) \\
&\forall (u,ng,d) \in ind(p_{min\_shutdown\_ramp}) \\
&\forall s \in stochastic\_path, \forall t \in time\_slice
\end{aligned}
$$

Upper bound on downward shut-down ramps

This constraint enforces an upper bound on the unit ramp during shutdown process. Usually, units will only provide shutdown ramps to non-reserve nodes. However, it is possible to include them, by adding them to the ramp defining node `ng`. The constraint is triggered by the existence of the max_shutdown_ramp parameter.

$$
\begin{aligned}
&+ \sum_{\substack{(u,n,d,s,t) \in shut\_down\_unit\_flow\_indices: \\ (u,n,d) \in (u,ng,d)}} v_{shut\_down\_unit\_flow}(u,n,d,s,t) \\
&<= \\
&+ \sum_{\substack{(u,s,t') \in units\_on\_indices: \\ (u,s) \in (u,s) \\ t' \in t\_overlap\_t(t)}} v_{units\_shut\_down}(u,s,t') \\
&\cdot p_{max\_shutdown\_ramp}(u,ng,d,s,t) \\
&\cdot p_{unit\_capacity}(u,ng,d,s,t) \\
&\cdot p_{conv\_cap\_to\_flow}(u,ng,d,s,t) \\
&\forall (u,ng,d) \in ind(p_{max\_shutdown\_ramp}) \\
&\forall s \in stochastic\_path, \forall t \in time\_slice
\end{aligned}
$$

Constraint on upward non-spinning shut-downs

For non-spinning downward reserves, online units can be scheduled for reserve provision through shut down if they have recovered their minimum up time. If nonspinning reserves are used the minimum up-time constraint becomes:

$$
\begin{aligned}
&v_{units\_on}(u,s,t)\\
&>= \sum_{\substack{(u,s,t')\in units\_on\_indices:\\ t'>t-p_{min\_up\_time}(u,s,t)}} v_{units\_started\_up}(u,s,t') \quad t'<=t\\
&+ \sum_{\substack{(u',n',s',t')\in nonspin\_units\_shut\_down\_indices:\\ (u',s',t')\in(u,s,t)}} v_{nonspin\_units\_shut\_down}(u',n',s',t')\\
&\forall(u,s,t)\in units\_on\_indices:\\
&u\in nonspin\_units\_started\_up\_indices
\end{aligned}
$$

Lower bound on the nonspinning downward reserve provision

A lower bound on the nonspinning reserve provision of a unit can be imposed by defining the min_res_shutdown_ramp parameter, which leads to the creation of the following constraint in the model:

$$
\begin{aligned}
&+ \sum_{\substack{(u,n,d,s,t)\in nonspin\_ramp\_down\_unit\_flow\_indices:\\ (u,n,d,s,t)\in(u,n,d,s,t)}} v_{nonspin\_ramp\_down\_unit\_flow}(u,n,d,s,t)\\
&<=\\
&+ \sum_{\substack{(u,n,s,t)\in nonspin\_units\_shut\_down\_indices:\\ (u,n,s,t)\in(u,n,s,t)}} v_{nonspin\_units\_shut\_down}(u,n,s,t)\\
&\cdot p_{min\_res\_shutdown\_ramp}(u,ng,d,s,t)\\
&\cdot p_{unit\_capacity}(u,ng,d,s,t)\\
&\cdot p_{conv\_cap\_to\_flow}(u,ng,d,s,t)\\
&\forall(u,ng,d)\in ind(p_{min\_res\_shutdown\_ramp})\\
&\forall s\in stochastic\_path, \forall t\in time\_slice
\end{aligned}
$$

Upper bound on the nonspinning downward reserve provision

An upper limit on the nonspinning reserve provision of a unit can be imposed by defining the max_res_shutdown_ramp parameter, which leads to the creation of the following constraint in the model:

$$
\begin{aligned}
&+ \sum_{\substack{(u,n,d,s,t)\in nonspin\_ramp\_down\_unit\_flow\_indices:\\ (u,n,d,s,t)\in(u,n,d,s,t)}} v_{nonspin\_ramp\_down\_unit\_flow}(u,n,d,s,t)\\
&<=\\
&+ \sum_{\substack{(u,n,s,t)\in nonspin\_units\_shut\_down\_indices:\\ (u,n,s,t)\in(u,n,s,t)}} v_{nonspin\_units\_shut\_down}(u,n,s,t)\\
&\cdot p_{max\_res\_shutdown\_ramp}(u,ng,d,s,t)\\
&\cdot p_{unit\_capacity}(u,ng,d,s,t)\\
&\cdot p_{conv\_cap\_to\_flow}(u,ng,d,s,t)\\
&\forall(u,ng,d)\in ind(p_{max\_res\_shutdown\_ramp})\\
&\forall s\in stochastic\_path, \forall t\in time\_slice
\end{aligned}
$$

Constraint on minimum node state for reserve provision

Storage nodes can also contribute to the provision of reserves. The amount of balancing contributions is limited by the ramps of the storage unit (see above) and by the node state:

$$
\begin{aligned}
&v_{node\_state}(n_{stor},s,t)\\
&>= p_{node\_state\_min}(n_{stor},s,t)\\
&+ \sum_{\substack{(u,n_{res},d,s,t)\in unit\_flow\_indices:\\ u\in unit\_flow\_indices; n=n_{stor})\\ p_{is\_reserve\_node}(n_{res})}} v_{unit\_flow}(u,n_{res},d,s,t)\\
&\cdot p_{minimum\_reserve\_activation\_time}(n_{res})\\
&\forall(n_{stor},s,t)\in node\_stochastic\_time\_indices: p_{has\_state}(n)
\end{aligned}
$$

Bounds on the unit capacity including ramping constraints

(Comment 2021-04-29: Currently under development)

Operating segments

Operating segments of units

The `unit_flow_op` operating segment variable is bounded by the difference between successive operating_points adjusted for unit_capacity

$$
\begin{aligned}
&v_{unit\_flow\_op}(u, n, op, s, t) \\
&<= p_{unit\_capacity}(u, n, d, s, t) \\
&\cdot v_{units\_available}(u, s, t) \\
&\cdot p_{unit\_conv\_cap\_to\_flow}(u, n, d, s, t) \\
&\cdot \Bigg(p_{operating\_points}(u, n, op, s, t) \\
&- \begin{cases} 0 & \text{if op} = 1 \\ p_{operating\_points}(u, n, op-1, s, t) & \text{otherwise} \end{cases} \Bigg) \\
&\forall (u, n, d, s, t) \in unit\_flow\_op\_indices
\end{aligned}
$$

Bounding unit flows by summing over operating segments

`unit_flow` is constrained to be the sum of all operating segment variables, `unit_flow_op`

$$
\begin{aligned}
&v_{unit\_flow}(u, n, s, t) \\
&= \sum_{op} v_{unit\_flow\_op}(u, n, op, s, t) \\
&\forall (u, n, d) \in operating\_point\_indices \\
&\forall (u, n, d, s, t) \in unit\_flow\_op\_indices
\end{aligned}
$$

Unit piecewise incremental heat rate

$$
\begin{aligned}
&v_{unit\_flow}(u, n_{in}, d, s, t) \\
&= \sum_{op} \Bigg(v_{unit\_flow\_op}(u, n_{out}, d, op, s, t) \\
&\qquad \cdot p_{unit\_incremental\_heat\_rate}(u, n_{in}, n_{out}, op, s, t) \Bigg) \\
&+ v_{units\_on}(u, s, t) \cdot p_{unit\_idle\_heat\_rate}(u, n_{in}, n_{out}, s, t) \\
&+ v_{units\_started\_up}(u, s, t) \cdot p_{unit\_start\_flow}(u, n_{in}, n_{out}, s, t) \\
&\forall (u, n_{in}, n_{out}, s, t) \in unit\_pw\_heat\_rate\_indices
\end{aligned}
$$

Bounds on commodity flows

Upper bound on cumulated unit flows

To impose a limit on the cumulative amount of certain commodity flows, a cumulative bound can be set by defining the parameter max_cum_in_unit_flow_bound for entire optimization window:

$$
\begin{aligned}
&\sum_{\substack{(u,n,d,s,t') \in unit\_flow\_indices: \\ (u,n,d,t') \in (ug,ng,d)}} v_{unit\_flow}(u, n, d, s, t') \cdot \Delta t' \\
&<= p_{max\_cum\_unit\_flow\_bound}(ug, ng, d, s, t) \\
&\forall (ug, ng, d) \in ind(p_{max\_cum\_unit\_flow\_bound})
\end{aligned}
$$

(Comment 2021-04-29: Currently under development)

Network constraints

Static constraints

Capacity constraint on connections

In a multi-commodity setting, there can be different commodities entering/leaving a certain connection. These can be energy-related commodities (e.g., electricity, natural gas, etc.), emissions, or other commodities (e.g., water, steel). The connection_capacity should be specified for at least one connection__to_node or connection__from_node relationship, in order to trigger a constraint on the maximum commodity flows to this location in each time step. When desirable, the capacity can be specified for a group of nodes (e.g. combined capacity for multiple products). Note that the conversion factor connection_conv_cap_to_flow has a default value of `1`, but can be adjusted in case the unit of measurement for the capacity is different to the connection flows unit of measurement.

$$
\begin{aligned}
&\sum_{\substack{(conn,n,d,s,t') \in connection\_flow\_indices: \\ (conn,n,d,s,t') \in (conn,ng,d,s,t)}} v_{connection\_flow}(conn,n,d,s,t') \cdot \Delta t' \\
&- \sum_{\substack{(conn,n,d_{reverse},s,t') \in connection\_flow\_indices: \\ (conn,n,s,t') \in (conn,ng,s,t) \\ d_{reverse} != d}} v_{connection\_flow}(conn,n,d_{reverse},s,t') \cdot \Delta t' \\
&<= p_{connection\_capacity}(conn,ng,d,s,t) \\
&\cdot p_{connection\_availability\_factor}(conn,s,t) \\
&\cdot p_{connection\_conv\_cap\_to\_flow}(conn,ng,d,s,t) \Delta t \\
&\forall (conn,ng,d) \in ind(p_{connection\_capacity}): \\
&\nexists p_{candidate\_connections}(conn) \\
&\forall t \in time\_slices, \\
&\forall s \in stochastic\_path
\end{aligned}
$$

If the connection is a candidate_connections, i.e. can be invested in, the connection capacity constraint translates to:

$$
\begin{aligned}
&\sum_{\substack{(conn,n,d,s,t') \in connection\_flow\_indices: \\ (conn,n,d,s,t') \in (conn,ng,d,s,t)}} v_{connection\_flow}(conn,n,d,s,t') \cdot \Delta t' \\
&- \sum_{\substack{(conn,n,d_{reverse},s,t') \in connection\_flow\_indices: \\ (conn,n,s,t') \in (conn,ng,s,t) \\ d_{reverse} != d}} v_{connection\_flow}(conn,n,d_{reverse},s,t') \cdot \Delta t' \\
&<= p_{connection\_capacity}(conn,ng,d,s,t) \\
&\cdot p_{connection\_availability\_factor}(conn,s,t) \\
&\cdot p_{connection\_conv\_cap\_to\_flow}(conn,ng,d,s,t) \Delta t \\
&\cdot \sum_{\substack{(conn,s,t') \in connections\_invested\_available\_indices: \\ (conn,s,t') \in (conn,s,t\_in\_t(t_{short}))}} v_{connections\_invest\_available(conn,s,t)} \qquad \forall (conn,ng,d) \in ind(p_{connection\_capacity}): \\
&\exists p_{candidate\_connections}(conn) \\
&\forall t \in time\_slices, \\
&\forall s \in stochastic\_path
\end{aligned}
$$

Fixed ratio between outgoing and incoming flows of a connection

By defining the parameters fix_ratio_out_in_connection_flow, max_ratio_out_in_connection_flow or min_ratio_out_in_connection_flow, a ratio can be set between **out**going and **in**coming flows from and to a connection.

In the most general form of the equation, two node groups are defined (an input node group ng_{in} and an output node group ng_{out}), and a linear relationship is expressed between both node groups. Note that whenever the relationship is specified between groups of multiple nodes, there remains a degree of freedom regarding the composition of the input node flows within group ng_{in} and the output node flows within group ng_{out}.

The constraint given below enforces a fixed, maximum or minimum ratio between outgoing and incoming connection_flow. Note that the potential node groups, that the parameters fix_ratio_out_in_connection_flow, max_ratio_out_in_connection_flow and min_ratio_out_in_connection_flow are defined on, are getting internally expanded to the members of the node group within the connection_flow_indices.

$$\sum_{\substack{(conn,n,d,s,t_{out}) \in connection\_flow\_indices: \\ (conn,n,d,s,t_{out}) \in (conn,ng_{out},:to\_node,s,t)}} v_{connection\_flow}(conn,n,d,s,t_{out}) \cdot \Delta t_{out}$$

$$\{$$

$$== p_{fix\_ratio\_out\_in\_connection\_flow}(conn, ng_{out}, ng_{in}, s, t),$$

$$<= p_{max\_ratio\_out\_in\_connection\_flow}(conn, ng_{out}, ng_{in}, s, t),$$

$$>= p_{min\_ratio\_out\_in\_connection\_flow}(conn, ng_{out}, ng_{in}, s, t)$$

$$\}$$

$$\cdot \sum_{\substack{(conn,n,d,s,t_{in}) \in connection\_flow\_indices: \\ (conn,n,d,s,t_{in}) \in (conn,ng_{in},:from\_node,s,t)}} v_{connection\_flow}(conn,n,d,s,t_{in}) \cdot \Delta t_{in}$$

$$\forall (conn, ng_{out}, ng_{in}) \in ind(p_{\{fix,max,min\}\_ratio\_out\_in\_connection\_flow}),$$

$$\forall t \in time\_slices, \forall s \in stochastic\_path$$

Specific network representation

In the following, the different specific network representations are introduced. While the Static constraints find application in any of the different networks, the following equations are specific to the discussed use cases. Currently, SpineOpt incorporated equations for pressure driven gas networks, nodal lossless DC power flows and PTDF based lossless DC power flow.

Pressure driven gas transfer

For gas pipelines it can be relevant a pressure driven gas transfer can be modelled, i.e. to account for linepack flexibility. Generally speaking, the main challenges related to pressure driven gas transfers are the non-convexities associated with the Weymouth equation. In SpineOpt, a convexified MILP representation has been implemented, which as been presented in Schwele - Coordination of Power and Natural Gas Systems: Convexification Approaches for Linepack Modeling. The approximation approach is based on the Taylor series expansion around fixed pressure points.

In addition to the already known variables, such as connection_flow and node_state, the start and end points of a gas pipeline connection are associated with the variable node_pressure. The variable is triggered by the has_pressure parameter. For more details on how to set up a gas pipeline, see also the advanced concept section on pressure driven gas transfer.

Maximum node pressure

In order to impose an upper limit on the maximum pressure at a node the maximum node pressure constraint can be included, by defining the parameter max_node_pressure which triggers the following constraint:

$$\sum_{\substack{(n,s,t') \in node\_pressure\_indices: \\ (n,s,t') \in (n,s,t)}} v_{node\_pressure}(n,s,t') \cdot \Delta t'$$

$$<= p_{max\_node\_pressure}(ng,s,t) \cdot \Delta t$$

$$\forall (ng) \in ind(p_{max\_node\_pressure}),$$

$$\forall t \in time\_slices,$$

$$\forall s \in stochastic\_path$$

As indicated in the equation, the parameter max_node_pressure can also be defined on a node group, in order to impose an upper limit on the aggregated node_pressure within one node group.

Minimum node pressure

In order to impose a lower limit on the pressure at a node the maximum node pressure constraint can be included, by defining the parameter min_node_pressure which triggers the following constraint:

$$\sum_{\substack{(n,s,t') \in node\_pressure\_indices: \\ (n,s,t') \in (n,s,t)}} v_{node\_pressure}(n,s,t') \cdot \Delta t'$$

$$>= p_{min\_node\_pressure}(ng,s,t) \cdot \Delta t$$

$$\forall (ng) \in ind(p_{min\_node\_pressure}),$$

$$\forall t \in time\_slices,$$

$$\forall s \in stochastic\_path$$

As indicated in the equation, the parameter min_node_pressure can also be defined on a node group, in order to impose a lower limit on the aggregated node_pressure within one node group.

Constraint on the pressure ratio between two nodes

If a compression station is located in between two nodes, the connection is considered to be active and a compression ratio between the two nodes can be imposed. The parameter compression_factor needs to be defined on a connection__node__node relationship, where the first node corresponds the origin node, before the compression, while the second node corresponds to the destination node, after compression. The existence of this parameter will trigger the following constraint:

$$\sum_{\substack{(n,s,t') \in node\_pressure\_indices: \\ (n,s,t') \in (ng2,s,t)}} v_{node\_pressure}(n,s,t') \cdot \Delta t'$$
$$<= p_{compression\_factor}(conn, ng1, ng2, s, t)$$
$$\sum_{\substack{(n,s,t') \in node\_pressure\_indices: \\ (n,s,t') \in (ng1,s,t)}} v_{node\_pressure}(n,s,t') \cdot \Delta t'$$
$$\forall (conn, ng1, ng2) \in ind(p_{compression\_factor}),$$
$$\forall t \in time\_slices,$$
$$\forall s \in stochastic\_path$$

Outer approximation through fixed pressure points

The Weymouth relates the average flows through a connection to the difference between the adjacent squared node pressures.

$$\begin{aligned}
&((v_{connection_flow}(conn, n_{orig}, :from_node, s, t) + v_{connection_flow}(conn, n_{dest}, :to_node, s, t))/2 \\
&- (v_{connection_flow}(conn, n_{dest}, :from_node, s, t) + v_{connection_flow}(conn, n_{orig}, :to_node, s, t))/2) \\
&\cdot \\
&|((v_{connection_flow}(conn, n_{orig}, :from_node, s, t) + v_{connection_flow}(conn, n_{dest}, :to_node, s, t))/2 \\
&- (v_{connection_flow}(conn, n_{dest}, :from_node, s, t) + v_{connection_flow}(conn, n_{orig}, :to_node, s, t))/2|) \\
&= K(conn) \cdot (v_{node_pressure}(n_{orig}, s, t)^2 - v_{node_pressure}(n_{dest}, s, t)^2)
\end{aligned}$$

Which can be rewritten as

$$\begin{aligned}
&((v_{connection_flow}(conn, n_{orig}, :from_node, s, t) + v_{connection_flow}(conn, n_{dest}, :to_node, s, t))/2 \\
&- (v_{connection_flow}(conn, n_{dest}, :from_node, s, t) + v_{connection_flow}(conn, n_{orig}, :to_node, s, t))/2) \\
&= \sqrt{K(conn) \cdot (v_{node_pressure}(n_{orig}, s, t)^2 - v_{node_pressure}(n_{dest}, s, t)^2)} \\
&\forall (v_{connection_flow}(conn, n_{orig}, :from_node, s, t) + v_{connection_flow}(conn, n_{dest}, :to_node, s, t))/2 > 0
\end{aligned}$$

and

$$\begin{aligned}
&((v_{connection_flow}(conn, n_{dest}, :from_node, s, t) + v_{connection_flow}(conn, n_{orig}, :to_node, s, t))/2 \\
&- (v_{connection_flow}(conn, n_{orig}, :from_node, s, t) + v_{connection_flow}(conn, n_{dest}, :to_node, s, t))/2) \\
&= \sqrt{K(conn) \cdot (v_{node_pressure}(n_{dest}, s, t)^2 - v_{node_pressure}(n_{orig}, s, t)^2)} \\
&\forall (v_{connection_flow}(conn, n_{orig}, :from_node, s, t) + v_{connection_flow}(conn, n_{dest}, :to_node, s, t))/2 < 0
\end{aligned}$$

where `K` corresponds to the natural gas flow constant.

The cone described by the Weymouth equation can be outer approximated by a number of tangent planes, using a set of fixed pressure points, as illustrated in Schwele - Integration of Electricity, Natural Gas and Heat Systems With Market-based Coordination. The bigM method is used to replace the sign function.

The linearized version of the Weymouth equation implemented in SpineOpt is given as follows:

$$
\begin{aligned}
&((v_{connection\_flow}(conn, n_{orig}, :from\_node, s, t) + v_{connection\_flow}(conn, n_{dest}, :to\_node, s, t))/2 \\
&<= p_{fixed\_pressure\_constant\_1}(conn, n_{orig}, n_{dest}, j, s, t) \cdot v_{node\_pressure}(n_{orig}, s, t) \\
&- p_{fixed\_pressure\_constant\_0}(conn, n_{orig}, n_{dest}, j, s, t) \cdot v_{node\_pressure}(n_{dest}, s, t) \\
&+ p_{big\_m} \cdot (1 - v_{binary\_gas\_connection\_flow}(conn, n_{dest}, :to\_node, s, t)) \\
&\forall (conn, n_{orig}, n_{dest}) \in ind(p_{fixed\_pressure\_constant\_1}) \\
&\forall j \in 1 : n(p_{fixed\_pressure\_constant\_1(connection=conn,node1=n_{orig},node2=n_dest)}) : \\
&p_{fixed\_pressure\_constant\_1}(conn, n_{orig}, n_{dest}, i = j)! = 0 \\
&\forall t \in time\_slices, \\
&\forall s \in stochastic\_path
\end{aligned}
$$

The parameters fixed_pressure_constant_1 and fixed_pressure_constant_0 should be defined in the database. For each considered fixed pressure point, they can be calculated as follows:

\[\begin{aligned} & p_{fixed_pressure_constant_1}(conn,n_{orig},n_{dest},j) \\ & = K(conn) \cdot p_{fixed_pressure}(n_{orig},j)/ \sqrt{p_{fixed_pressure}(n_{orig},j)^2 - p_{fixed_pressure}(n_{dest},j)^2\\ & p_{fixed_pressure_constant_0}(conn,n_{orig},n_{dest},j) \\ & = K(conn) \cdot p_{fixed_pressure}(n_{dest},j)/ \sqrt{p_{fixed_pressure}(n_{orig},j)^2 - p_{fixed_pressure}(n_{dest},j)^2\\ \end{aligned}\] where K corrsponds to the natural gas flow constant.

The big_m parameter combined with the variable binary_gas_connection_flow together with the equations on unitary gas flow and on the maximum gas flow ensure that the bound on the average flow through the fixed pressure points becomes active, if the flow is in a positive direction for the observed set of connection, node1 and node2.

Enforcing unidirectional flow

As stated above, the flow through a connection can only be in one direction at at time. Whether a flow is active in a certain direction is indicated by the binary_gas_connection_flow variable, which takes a value of `1` if the direction of flow is positive. To ensure that the binary_gas_connection_flow in the opposite direction then takes the value `0`, the following constraint is enforced:

$$
\begin{aligned}
&v_{binary\_gas\_connection\_flow}(conn, n_{orig}, :to\_node, s, t)) \\
&= (1 - v_{binary\_gas\_connection\_flow}(conn, n_{dest}, :to\_node, s, t)) \\
&\forall (n, d, s, t) \in binary\_gas\_connection\_flow\_indices
\end{aligned}
$$

Gas connection flow capacity

To enforce that the averge flow of a connection is only in one direction, the flow in the opposite direction is forced to be `0` by the following euqation. For the connection flow in the direction of flow the parameter big_m should be chosen large enough to not become binding.

$$
\begin{aligned}
&((v_{connection\_flow}(conn, n_{orig}, :from\_node, s, t) + v_{connection\_flow}(conn, n_{dest}, :to\_node, s, t))/2 \\
&<= p_{big\_m} \cdot v_{binary\_gas\_connection\_flow}(conn, n_{dest}, :to\_node, s, t) \\
&\forall (conn, n_{orig}, n_{dest}) \in ind(p_{fixed\_pressure\_constant\_1}) \\
&\forall t \in time\_slices, \\
&\forall s \in stochastic\_path
\end{aligned}
$$

Linepack storage flexibility

In order to account for linepack flexibility, i.e. storage capability of a connection, the linepack storage is linked to the average pressure of the adjacent nodes by the following equation, triggered by the parameter connection_linepack_constant:

$$
\begin{aligned}
&v_{node\_state}(n_{stor}, s, t)\Delta t \\
&= p_{connection\_linepack\_constant}(conn, n_{stor}, n_{ngroup})/2 \sum_{\substack{(n,s,t') \in node\_pressure\_indices: \\ (n,s,t') \in (ng,s,t)}} v_{node\_pressure}(n, s, t') \cdot \Delta t' \\
&\forall (conn, n_{stor}, n_{ngroup}) \in ind(p_{connection\_linepack\_constant}) \\
&\forall t \in time\_slices, \\
&\forall s \in stochastic\_path
\end{aligned}
$$

Note that the parameter connection_linepack_constant should be defined on a connection__node__node relationship, where the first node corresponds to the linepack storage node, whereas the second node corresponds to the node group of both start and end nodes of the pipeline.

Nodebased lossless DC power flow

For the implementation of the nodebased loss DC powerflow model, a new variable node_voltage_angle is introduced. See also has_voltage_angle. For further explanation on setting up a database for nodal lossless DC power flow, see the advanced concept chapter on Lossless nodal DC power flows.

Maximum node voltage angle

In order to impose an upper limit on the maximum voltage angle at a node the maximum node voltage angle constraint can be included, by defining the parameter max_voltage_angle which triggers the following constraint:

$$\sum_{\substack{(n,s,t') \in node\_voltage\_angle\_indices: \\ (n,s,t') \in (n,s,t)}} v_{node\_voltage\_angle}(n,s,t') \cdot \Delta t' <= p_{max\_voltage\_angle}(ng,s,t) \cdot \Delta t \quad \forall (ng) \in ind(p_{max\_voltage\_angle}), \forall t \in time\_slices, \forall s \in stochastic\_path$$

As indicated in the equation, the parameter max_voltage_angle can also be defined on a node group, in order to impose an upper limit on the aggregated node_voltage_angle within one node group.

Minimum node voltage angle

In order to impose a lower limit on the voltage angle at a node the maximum node voltage angle constraint can be included, by defining the parameter min_voltage_angle which triggers the following constraint:

$$\sum_{\substack{(n,s,t') \in node\_voltage\_angle\_indices: \\ (n,s,t') \in (n,s,t)}} v_{node\_voltage\_angle}(n,s,t') \cdot \Delta t' >= p_{min\_voltage\_angle}(ng,s,t) \cdot \Delta t \quad \forall (ng) \in ind(p_{min\_voltage\_angle}), \forall t \in time\_slices, \forall s \in stochastic\_path$$

As indicated in the equation, the parameter min_voltage_angle can also be defined on a node group, in order to impose a lower limit on the aggregated node_voltage_angle within one node group.

Voltage angle to connection flows

To link the flow over a connection to the voltage angles of the adjacent nodes, the following constraint is imposed. Note that this constraint is only generated if the parameter connection_reactance is defined for a connection__node__node relationship and if a fix_ratio_out_in_connection_flow is defined for the corresponding connection, node, node tuples.

$$
\begin{aligned}
&+ \sum_{\substack{(conn,n',d,s,t) \in connection\_flow\_indices: \\ d_{from} == :from\_node \\ n' \in n_{from}}} v_{connection\_flow}(conn, n', d, s, t) \\
&- \sum_{\substack{(conn,n',d,s,t) \in connection\_flow\_indices: \\ d_{from} == :from\_node \\ n' \in n_{to}}} v_{connection\_flow}(conn, n', s, t) \\
&= \\
&1/p_{connection\_reactance}(conn) \cdot p_{connection\_reactance\_base}(conn) \\
&\cdot \Big(\sum_{\substack{(n,s,t') \in node\_voltage\_angle\_indices: \\ (n,s,t') \in (n_{from},s,t)}} v_{node\_voltage\_angle}(n, s, t') \cdot \Delta t' \\
&- \sum_{\substack{(n,s,t') \in node\_voltage\_angle\_indices: \\ (n,s,t') \in (n_{to},s,t)}} v_{node\_voltage\_angle}(n, s, t') \cdot \Delta t' \Big) \\
&(conn, n_{to}, n_{from}) \in indices(p_{fix\_ratio_{out}in_{c}onnection_{f}low}) \\
&\forall t \in time\_slices, \\
&\forall s \in stochastic\_path
\end{aligned}
$$

PTDF based DC lossless powerflow

Connection intact flow PTDF

The power transfer distribution factors are a property of the network reactances. `ptdf(n, c)` represents the fraction of an injection at node n that will flow on connection c. The flow on connection c is then the sum over all nodes of `ptdf(n, c)*net_injection(c)`. connection_intact_flow represents the flow on each line of the network will all candidate connections with PTDF-based flow present in the network.

$$
\begin{aligned}
&+ v_{connection\_intact\_flow}(c, n_{to}, d_{to}, s, t) \\
&- v_{connection\_intact\_flow}(c, n_{to}, d_{from}, s, t) \\
&== \sum_{n_{inj}} \Big(v_{node\_injection}(n_{inj}, s, t) \cdot p_{ptdf}(c, n_{inj}) \Big) \\
&\forall (c, n_{to}, s, t) \in connection\_ptdf\_flow\_indices
\end{aligned}
$$

N-1 post contingency connection flow limits

The N-1 security constraint for the post-contingency flow on monitored connection, `c_mon`, upon the outage of contingency connection, `c_conn`, is formed using line outage distribution factors (lodf). `lodf(c_con, c_mon)` represents the fraction of the pre-contingency flow on connection `c_conn` that will flow on `c_mon` if `c_conn` is disconnected. If connection `c_conn` is disconnected, the post-contingency flow on monitored connection connection `c_mon` is the pre-contingency `connection_flow` on `c_mon` plus the line outage distribution factor (`lodf`) times the pre-contingency `connection_flow` on `c_conn`. This post-contingency flow should be less than the connection_emergency_capacity of `c_mon`.

$$
\begin{aligned}
&+ v_{connection\_flow}(c_{mon}, n_{mon\_to}, d_{to}, s, t) \\
&- v_{connection\_flow}(c_{mon}, n_{mon\_to}, d_{from}, s, t) \\
&+ p_{lodf}(c_{conn}, c_{mon}) \cdot \Big(\\
&\quad + v_{connection\_flow}(c_{conn}, n_{conn\_to}, d_{to}, s, t) \\
&\quad - v_{connection\_flow}(c_{conn}, n_{conn\_to}, d_{from}, s, t) \Big) \\
&< min(p_{connection\_emergency\_capacity}(c_{mon}, n_{conn\_to}, d_{to}, s, t), p_{connection\_emergency\_capacity}(c_{mon}, n_{conn\_to}, d_{from}, s, t)) \\
&\forall (c_{mon}, c_{conn}, s, t) \in constraint\_connection\_flow\_lodf\_indices
\end{aligned}
$$

Investments

Investments in units

Economic lifetime of a unit

Enforces the minimum duration of a `unit`'s investment decision. Once a `unit` has been invested-in, it must remain invested-in for `unit_investment_lifetime`.

$$v_{units\_invested\_available}(u, s, t) >= \sum_{\substack{(u,s,t') \in units\_invested\_available\_indices: \\ t' >= t - p_{unit\_investment\_lifetime}(u,s,t), \\ t' <= t}} v_{units\_invested}(u, s, t') \quad \forall (u, s, t) \in unit\_investment\_lifetime\_indices$$

Technical lifetime of a unit

(Comment 2021-04-29: Currently under development)

Available Investment Units

The number of available invested-in units at any point in time is less than the number of investment candidate units.

$$v_{units\_invested\_available}(u, s, t) < p_{candidate\_units}(u, s, t) \quad \forall u \in candidate\_units\_indices, \quad \forall (u, s, t) \in units\_invested\_available\_indices$$

Investment transfer

`units_invested` represents the point-in-time decision to invest in a unit or not while `units_invested_available` represents the invested-in units that are available in a specific timeslice. This constraint enforces the relationship between `units_invested`, `units_invested_available` and `units_mothballed` in adjacent timeslices.

$$v_{units\_invested\_available}(u, s, t_{after}) - v_{units\_invested}(u, s, t_{after}) + v_{units\_monthballed}(u, s, t_{after}) == v_{units\_invested\_available}(u, s, t_{before}) \quad \forall (u, s, t_{after}) \in units\_invested\_available\_indices, \quad \forall t_{before} \in t\_before\_t(t\_after = t_{after}) : t_{before} \in units\_invested\_available\_indices$$

Investments in connections

Available invested-in connections

The number of available invested-in connections at any point in time is less than the number of investment candidate connections.

$$v_{connections\_invested\_available}(c, s, t) < p_{candidate\_connections}(c, s, t) \quad \forall c \in candidate\_connections\_indices, \quad \forall (c, s, t) \in connections\_invested\_available\_indices$$

Transfer of previous investments

`connections_invested` represents the point-in-time decision to invest in a connection or not while `connections_invested_available` represents the invested-in connections that are available in a specific timeslice. This constraint enforces the relationship between `connections_invested`, `connections_invested_available` and `connections_decommissioned` in adjacent timeslices.

$$
\begin{aligned}
& v_{connections\_invested\_available}(c, s, t_{after}) \\
& - v_{connections\_invested}(c, s, t_{after}) \\
& + v_{connections\_decommissioned}(c, s, t_{after}) \\
& == v_{connections\_invested\_available}(c, s, t_{before}) \\
& \forall (c, s, t_{after}) \in connections\_invested\_available\_indices, \\
& \forall t_{before} \in t\_before\_t(t\_after = t_{after}) : t_{before} \in connections\_invested\_available\_indices
\end{aligned}
$$

Intact network ptdf-based flows on connections

Enforces the relationship between connection_intact_flow (flow with all investments assumed in force) and connection_flow connection_intact_flow is the flow on all lines with all investments assumed in place. This constraint ensures that the connection_flow is connection_intact_flow plus additional flow contributions from investment connections that are not invested in.

$$
\begin{aligned}
& + v_{connection\_flow}(c, n_{to}, d_{from}, s, t) \\
& - v_{connection\_flow}(c, n_{to}, d_{to}, s, t) \\
& - v_{connection\_intact\_flow}(c, n_{to}, d_{from}, s, t) \\
& + v_{connection\_intact\_flow}(c, n_{to}, d_{to}, s, t) \\
& == \\
& \sum_{c_{candidate}, n_{to_candidate}} p_{lodf}(c_{candidate}, c) \cdot \Big(\\
& \qquad + v_{connection\_flow}(c_{candidate}, n_{to_candidate}, d_{from}, s, t) \\
& \qquad - v_{connection\_flow}(c_{candidate}, n_{to_candidate}, d_{to}, s, t) \\
& \qquad - v_{connection\_intact\_flow}(c_{candidate}, n_{to_candidate}, d_{from}, s, t) \\
& \qquad + v_{connection\_intact\_flow}(c_{candidate}, n_{to_candidate}, d_{to}, s, t) \Big) \\
& \forall (c, n_{to}, s, t) \in connection\_flow\_intact\_flow\_indices
\end{aligned}
$$

Intact connection flows capacity

Similarly to constraint_connection_flow_capacity, limits connection_intact_flow according to connection_capacity

$$
\begin{aligned}
& \sum_{\substack{(conn,n,d,s,t') \in connection\_intact\_flow\_indices: \\ (conn,n,d,s,t') \in (conn,ng,d,s,t)}} v_{connection\_intact\_flow}(conn, n, d, s, t') \cdot \Delta t' \\
& - \sum_{\substack{(conn,n,d_{reverse},s,t') \in connection\_intact\_flow\_indices: \\ (conn,n,s,t') \in (conn,ng,s,t) \\ d_{reverse} != d}} v_{connection\_intact\_flow}(conn, n, d_{reverse}, s, t') \cdot \Delta t' \\
& <= p_{connection\_capacity}(conn, ng, d, s, t) \\
& \cdot p_{connection\_availability\_factor}(conn, s, t) \\
& \cdot p_{connection\_conv\_cap\_to\_flow}(conn, ng, d, s, t) \Delta t \\
& \forall (conn, ng, d) \in ind(p_{connection\_capacity}) : \\
& \forall t \in time\_slices, \\
& \forall s \in stochastic\_path
\end{aligned}
$$

Fixed ratio between outgoing and incoming intact flows of a connection

For ptdf-based lossless DC power flow, ensures that the output flow to the `to_node` equals the input flow from the `from_node`.

$$
\begin{aligned}
& + v_{connection\_intact\_flow}(c, n_{out}, d_{to}, s, t) \\
& == \\
& + v_{connection\_intact\_flow}(c, n_{in}, d_{from}, s, t) \\
& \forall (c, n_{in}, n_{out}, s, t) \in connection\_intact\_flow\_indices
\end{aligned}
$$

Lower bound on candidate connection flow

For candidate connections with PTDF-based poweflow, together with constraint_candidate_connection_flow_ub, this constraint ensures that connection_flow is zero if the candidate connection is not invested-in and equals connection_intact_flow otherwise.

$$
\begin{aligned}
&+ v_{connection\_flow}(c, n, d, s, t) \\
&>= \\
&+ v_{connection\_intact\_flow}(c, n, d, s, t) \\
&- p_{connection\_capacity}(c, n, d, s, t) \cdot (p_{candidate\_connections}(c, s, t) - v_{connections\_invested\_available}(c, s, t)) \\
&\forall (c, n, d, s, t) \in constraint\_candidate\_connection\_flow\_lb\_indices
\end{aligned}
$$

Upper bound on candidate connection flow

For candidate connections with PTDF-based poweflow, together with constraint_candidate_connection_flow_lb, this constraint ensures that connection_flow is zero if the candidate connection is not invested-in and equals connection_intact_flow otherwise.

$$
\begin{aligned}
&+ v_{connection\_flow}(c, n, d, s, t) \\
&<= \\
&+ v_{connection\_intact\_flow}(c, n, d, s, t) \\
\\
&\forall (c, n, d, s, t) \in constraint\_candidate\_connection\_flow\_ub\_indices
\end{aligned}
$$

Economic lifetime of a connection

Enforces the minimum duration of a `connection`'s investment decision. Once a `connection` has been invested-in, it must remain invested-in for `connection_investment_lifetime`.

$$
\begin{aligned}
&v_{connections\_invested\_available}(c, s, t) \\
&>= \sum_{\substack{(c,s,t') \in connections\_invested\_available\_indices: \\ t' >= t - p_{connection\_investment\_lifetime}(c,s,t), \\ t' <= t}} v_{connections\_invested}(c, s, t') \\
&\forall (c, s, t) \in connection\_investment\_lifetime\_indices
\end{aligned}
$$

Technical lifetime of a connection

(Comment 2021-04-29: Currently under development)

Investments in storages

Note: can we actually invest in nodes that are not storages? (e.g. new location)

Available invested storages

The number of available invested-in storages at node n at any point in time is less than the number of investment candidate storages at that node.

$$
\begin{aligned}
&v_{storages\_invested\_available}(n, s, t) \\
&< p_{candidate\_storages}(n, s, t) \\
&\forall (n) \in candidate\_storages\_indices, \\
&\forall (n, s, t) \in storages\_invested\_available\_indices
\end{aligned}
$$

Storage capacity transfer?

`storages_invested` represents the point-in-time decision to invest in storage at a node, n or not while `storages_invested_available` represents the invested-in storages that are available at a node in a specific timeslice. This constraint enforces the relationship between `storages_invested`, `storages_invested_available` and `storages_decommissioned` in adjacent timeslices.

$$
\begin{aligned}
&v_{storages\_invested\_available}(n, s, t_{after}) \\
&- v_{storages\_invested}(n, s, t_{after}) \\
&+ v_{storages\_decommissioned}(n, s, t_{after}) \\
&== v_{storages\_invested\_available}(n, s, t_{before}) \\
&\forall (n, s, t_{after}) \in storages\_invested\_available\_indices, \\
&\forall t_{before} \in t\_before\_t(t\_after = t_{after}) : t_{before} \in storages\_invested\_available\_indices
\end{aligned}
$$

Economic lifetime of a storage

Enforces the minimum duration of a `storage` investment decision at node n. Once a `storage` has been invested-in, it must remain invested-in for `storage_investment_lifetime`.

$$
\begin{aligned}
& v_{storages\_invested\_available}(n,s,t) \\
& >= \sum_{\substack{(n,s,t') \in storages\_invested\_available\_indices: \\ t'>=t-p_{storage\_investment\_lifetime}(n,s,t), \\ t'<=t}} v_{storages\_invested}(n,s,t') \\
& \forall (n,s,t) \in storage\_investment\_lifetime\_indices
\end{aligned}
$$

Technical lifetime of a storage

(Comment 2021-04-29: Currently under development)

Capacity transfer

(Comment 2021-04-29: Currently under development)

Early retirement of capacity

(Comment 2021-04-29: Currently under development)

Benders decomposition

This section describes the high-level formulation of the benders-decomposed problem.

Taking the simple example of minimising capacity and operating cost for a fleet of units with a linear cost coefficient '$operational\_cost_u$' :

$$
\begin{aligned}
& minimise: \\
& \quad + \sum_{u} p_{unit\_investment\_cost}(u) \cdot v_{units\_invested}(u) \\
& \quad + \sum_{u,n,t} p_{operational\_cost} \cdot v_{unit\_flow}(u,n,t) \\
& subject\ to: \\
& \quad flow_{u,n,t} \leq p_{unit\_capacity}(u,n,t) \cdot (v_{units\_available} + v_{units\_invested\_available}(u,n,t)) \\
& \quad \sum_{u,n,t} v_{unit\_flow}(u,t) = p_{demand}(n,t)
\end{aligned}
$$

So this is a single problem that can't be decoupled over `t` because the investment variables `units_invested_available` couple the timesteps together. If `units_invested_available` were a constant in the problem, then all `t`'s could be solved individually. This is the basic idea in Benders decomposition. We decompose the problem into a master problem and sub problems with the master problem optimising the coupling investment variables which are treated as constants in the sub problems.

The master problem in the initial benders iteration is simply to minimise total investment costs:

$$
\begin{aligned}
& minimise Z: \\
& \quad Z \geq \sum_{u} p_{unit\_investment\_cost}(u) \cdot v_{units\_invested}(u)
\end{aligned}
$$

The solution to this problem yields values for the investment variables which are fixed as '$\overline{units\_invested_u}$' in the sub problem and will be zero in the first iteration.

The sub problem for benders iteration `b` then becomes :

$$
\begin{aligned}
&minimise: \\
&\quad obj_b = + \sum_{u,n,t} p_{operational\_cost}(u) \cdot v_{unit\_flow}(u,n,t) \\
&subject\ to: \\
&\quad v_{unit_flow}(u,n,t) \leq p_{unit\_capacity}(u) \cdot (v_{units\_available}(u,t) + p_{units\_invested\_available}(u,t)) \qquad \mu_{b,u,t} \\
&\quad \sum_{u,n,t} v_{unit_flow}(u,n,t) = p_{demand}(n,t)
\end{aligned}
$$

This sub problem can be solved individually for each `t`. This is pretty trivial in this small example but if we consider a single t to be a single rolling horizon instead, decoupling the investment variables means that each rolling horizon can be solved individually rather than having to solve the entire model horizon as a single problem.

$$
`\mu_{u,t}`
$$

is the marginal value of the capacity constraint and can be interpreted as the decrease in the objective function at time `t` for an additional MW of flow from unit `u`. This information is used to construct a benders cut which represents the reduction in the sub problem objective function which is possible in this benders iteration by adjusting the variable units_investment. This is effectively the decrease in operating costs possible by adding another unit of type `u` and is expressed as :

$$
`obj_b + \sum_{u,t} p_{unit\_capacity}(u,n,t) \cdot \mu_{b,u,t} \cdot (v_{units\_invested}(u,t) - p_{units\_invested}(u,b,t))`
$$

In the first benders iteration, the value of the investment variables will have been zero so $`p_{units\_invested}(u,b,t)`$ will have the value of zero and thus the expression represents the total reduction in cost from an addition of a new unit of type `u`. This Benders cut is added to the master problem which then becomes, for each subsequent benders iteration, b:

$$
\begin{aligned}
&minimise Z: \\
&\quad Z \geq \sum_{u,t} p_{unit\_investment\_cost}(u) \cdot v_{units\_invested}(u,t) \\
&subject\ to: \\
&\quad Z \geq + \sum_{u} p_{unit\_investment\_cost}(u) \cdot v_{units\_invested}(u,t) \\
&\qquad + \sum_{u,t} p_{unit\_capacity}(u,t) \cdot \mu_{b,u,t} \cdot (v_{units\_invested}(u,t) - p_{units\_invested}(u,b,t)) \qquad \forall b
\end{aligned}
$$

Note the benders cuts are added as inequalities because they represent an upper bound on the value we are going to get from adjusting the master problem variables in that benders iteration. If we consider the example of renewable generation - because it's marginal cost is zero, on the first benders iteration, it could look like there would be a lot of value in increasing the capacity because of the marginal values from the sub problems. However, when the capacity variables are increased accordingly and curtailment occurs in the sub-problems, the marginal values will be zero when curtailment occurs and so, other resources may become optimal in subsequent iterations.

This is a simple example but it illustrates the general strategy. The algorithm pseudo code looks something like this:

```
initialise master problem
initialise sub problem
solve first master problem
create master problem variable time series
solve rolling spine opt model
save zipped marginal values
while master problem not converged
    update master problem
    solve master problem
    update master problem variable timeseries for benders iteration b
    rewind sub problem
    update sub problem
    solve rolling spine opt model
    save zipped marginal values
    test for convergence
end
```

Benders cuts

The benders cuts for the problem including all investments in candidate connections, storages and units is given below.

$$
\begin{aligned}
& v_{objective\_lower\_bound}(b) \\
& >= \\
& + \sum_{u,s,t} p_{units\_invested\_available\_mv}(u,t,b) \cdot [v_{units\_invested\_available}(u,s,t) - p_{units\_invested\_available\_bi}(u,t,b)] \\
& + \sum_{c,s,t} p_{connections\_invested\_available\_mv}(c,t,b) \cdot [v_{connections\_invested\_available}(c,s,t) - p_{connections\_invested\_available\_bi}(c,t,b)] \\
& + \sum_{n,s,t} p_{storages\_invested\_available\_mv}(n,t,b) \cdot [v_{storages\_invested\_available}(n,s,t) - p_{storages\_invested\_available\_bi}(n,t,b)]
\end{aligned}
$$

where

$$
`p_{units\_invested\_available\_mv}`
$$

is the reduced cost of the units_invested_available fixed sub-problem variable, representing the reduction in operating costs possible from an investment in a unit of this type, `$p_{connections\_invested\_available\_mv}$` is the reduced cost of the connections_invested_available fixed sub-problem variable, representing the reduction in operating costs possible from an investment in a connection of this type, `$p_{storages\_invested\_available\_mv}$` is the reduced cost of the storages_invested_available fixed sub-problem variable, representing the reduction in operating costs possible from an investment in a `storage` of this type, `$p_{units\_invested\_available\_bi}(u,t,b)$` is the value of the fixed sub problem variable units_invested_available(u,t) in benders iteration `b`, `$p_{connections\_invested\_available\_bi}(c,t,b)$` is the value of the fixed sub problem variable connections_invested_available(c,t) in benders iteration `b` and `$p_{storages\_invested\_available\_bi}(n,t,b)$` is the value of the fixed sub problem variable storages_invested_available(n,t) in benders iteration `b`

User constraints

Unit constraint

The unit_constraint is a generic data-driven custom constraint, which allows for defining constraints involving multiple units, nodes, or connections. The constraint_sense parameter changes the sense of the unit_constraint, while the right_hand_side parameter allows for defining the constant terms of the constraint.

Coefficients for the different variables appearing in the unit_constraint are defined using relationships, like e.g. unit__from_node__unit_constraint and connection__to_node__unit_constraint for unit_flow and connection_flow variables, or unit__unit_constraint and node__unit_constraint for units_on, units_started_up, and node_state variables.

For more information, see the dedicated article on Unit Constraints

$$
\begin{aligned}
&+ \sum_{u,n \in unit\_\_node\_\_unit\_constraint(uc),t,s} \\
&\begin{cases}
\sum_{op} v_{unit\_flow\_op}(u,n,d,op,s,t) \cdot p_{unit\_flow\_coefficient}(u,n,op,uc,s,t) & \text{if } |operating\_points(u)| > 1 \\
v_{unit\_flow}(u,n,d,s,t) \cdot p_{unit\_flow\_coefficient}(u,n,uc,s,t) & \text{otherwise}
\end{cases} \\
&+ \sum_{u \in unit\_\_unit\_constraint(uc),t,s} v_{units\_started\_up}(u,s,t) \cdot p_{units\_started\_up\_coefficient}(u,uc,s,t) \\
&+ \sum_{u \in unit\_\_unit\_constraint(uc),t,s} v_{units\_on}(u,s,t) \cdot p_{units\_on\_coefficient}(u,uc,s,t) \\
&+ \sum_{c,n \in connection\_\_node\_\_unit\_constraint(uc),t,s} v_{connection\_flow}(c,n,d,s,t) \cdot p_{connection\_flow\_coefficient}(c,n,uc,s,t) \\
&+ \sum_{n \in node\_\_unit\_constraint(uc),t,s} v_{node\_state}(n,s,t) \cdot p_{node\_state\_coefficient}(n,uc,s,t) \\
&+ \sum_{n \in node\_\_unit\_constraint(uc),t,s} p_{demand}(n,s,t) \cdot p_{demand\_coefficient}(n,uc,s,t) \\
&\begin{cases}
== & \text{if } p_{constraint\_sense}(uc) = \text{"=="} \\
>= & \text{if } p_{constraint\_sense}(uc) = \text{">="} \\
<= & \text{otherwise}
\end{cases} \\
&+ p_{right\_hand\_side}(uc,t,s) \\
&\forall uc,t,s \in constraint\_unit\_constraint\_indices
\end{aligned}
$$

« Variables

Objective »

Objective function

The objective function of SpineOpt expresses the minimization of the total system costs associated with maintaining and operating the considered energy system.

$$\begin{aligned}\min obj = & v_{unit\_investment\_costs} + v_{connection\_investment\_costs} + v_{storage\_investment\_costs} \\ & + v_{fixed\_om\_costs} + v_{variable\_om\_costs} + v_{fuel\_costs} + v_{start\_up\_costs} \\ & + v_{shut\_down\_costs} + v_{ramp\_costs} + v_{res\_proc\_costs} \\ & + v_{renewable\_curtailment\_costs} + v_{connection\_flow\_costs} + v_{taxes} + v_{objective\_penalties}\end{aligned}$$

Note that each cost term is reflected here as a separate variable that can be expressed mathematically by the equations below. All cost terms are weighted by the associated scenario and temporal block weights. To enhance readability and avoid writing a product of weights in every cost term, all weights are combined in a single weight parameter $p_{weight}(\dots)$. As such, the indices associated with each weight parameter indicate which weights are included.

Unit investment costs

To take into account unit investments in the objective function, the parameter unit_investment_cost can be defined. For all tuples of (unit, scenario, timestep) in the set `units_invested_available_indices` for which this parameter is defined, an investment cost term is added to the objective function if a unit is invested in during the current optimization window. The total unit investment costs can be expressed as:

$$\begin{aligned}&v_{unit\_investment\_costs} \\ &= \sum_{\substack{(u,s,t) \in units\_invested\_available\_indices: \\ u \in ind(p_{unit\_investment\_cost})}} v_{units\_invested}(u,s,t) \cdot p_{unit\_investment\_cost}(u,s,t) \cdot p_{weight}(u,s,t)\end{aligned}$$

Connection investment costs

To take into account connection investments in the objective function, the parameter connection_investment_cost can be defined. For all tuples of (connection, scenario, timestep) in the set `connections_invested_available_indices` for which this parameter is defined, an investment cost term is added to the objective function if a connection is invested in during the current optimization window. The total connection investment costs can be expressed as:

$$\begin{aligned}&v_{connection\_investment\_costs} \\ &= \sum_{\substack{(conn,s,t) \in connections\_invested\_available\_indices: \\ conn \in ind(p_{connection\_investment\_cost})}} v_{connections\_invested}(conn,s,t) \cdot p_{connection\_investment\_cost}(conn,s,t) \cdot p_{weight}(conn,s,t)\end{aligned}$$

Storage investment costs

To take into account storage investments in the objective function, the parameter storage_investment_cost can be defined. For all tuples of (node, scenario, timestep) in the set `storages_invested_available_indices` for which this parameter is defined, an investment cost term is added to the objective function if a node storage is invested in during the current optimization window. The total storage investment costs can be expressed as:

$$\begin{aligned}&v_{storage\_investment\_costs} \\ &= \sum_{\substack{(n,s,t) \in storages\_invested\_available\_indices: \\ n \in ind(p_{storage\_investment\_cost})}} v_{storages\_invested}(n,s,t) \cdot p_{storage\_investment\_cost}(n,s,t) \cdot p_{weight}(n,s,t)\end{aligned}$$

Fixed O&M costs

Fixed operation and maintenance costs associated with a specific unit can be accounted for by defining the parameters fom_cost and unit_capacity. For all tuples of (unit, {node,node_group}, direction) for which these parameters are defined, and for which tuples (unit, scenario, timestep) exist in the set `units_on_indices`, a fixed O&M cost term is added to the objective function. Note that, as the `units_on_indices` are used to retrieve the relevant time slices, the unit of the fom_cost parameter should be given per resolution of the units_on. The total fixed O&M costs can be expressed as:

$$v_{fixed\_om\_costs} = \sum_{\substack{(u,n,d) \in ind(p_{unit\_capacity}): \\ u \in ind(p_{fom\_cost})}} \sum_{(u,s,t) \in units\_on\_indices} p_{unit\_capacity}(u,n,d,s,t) \cdot p_{number\_of\_units}(u,s,t) \cdot p_{fom\_cost}(u,s,t) \cdot p_{weight}(t) \cdot p_{duration}(t)$$

Variable O&M costs

Variable operation and maintenance costs associated with a specific unit can be accounted for by defining the parameter (vom_cost). For all tuples of (unit, {node,node_group}, direction, scenario, timestep) in the set `unit_flow_indices` for which this parameter is defined, a variable O&M cost term is added to the objective function. As the parameter vom_cost is a dynamic parameter, the cost term is multiplied with the duration of each timestep. The total variable O&M costs can be expressed as:

$$v_{variable\_om\_costs} = \sum_{\substack{(u,n,d,s,t) \in unit\_flow\_indices: \\ (u,n,d) \in ind(p_{vom\_cost})}} v_{unit\_flow}(u,n,d,s,t) \cdot p_{vom\_cost}(u,n,d,s,t) \cdot p_{weight}(n,s,t) \cdot p_{duration}(t)$$

Fuel costs

Fuel costs associated with a specific unit can be accounted for by defining the parameter fuel_cost. For all tuples of (unit, {node,node_group}, direction, scenario, timestep) in the set `unit_flow_indices` for which this parameter is defined, a fuel cost term is added to the objective function. As the parameter fuel_cost is a dynamic parameter, the cost term is multiplied with the duration of each timestep. The total fuel costs can be expressed as:

$$v_{fuel\_costs} = \sum_{\substack{(u,n,d,s,t) \in unit\_flow\_indices: \\ (u,n,d) \in ind(p_{fuel\_cost})}} v_{unit\_flow}(u,n,d,s,t) \cdot p_{fuel\_cost}(u,n,d,s,t) \cdot p_{weight}(n,s,t) \cdot p_{duration}(t)$$

Connection flow costs

To account for operational costs associated with flows over a specific connection, the connection_flow_cost parameter can be defined. For all tuples of (conn, {node,node_group}, direction, scenario, timestep) in the set `connection_flow_indices` for which this parameter is defined, a connection flow cost term is added to the objective function. The total connection flow costs can be expressed as:

$$v_{connection\_flow\_costs} = \sum_{\substack{(conn,n,d,s,t) \in connection\_flow\_indices: \\ conn \in ind(p_{connection\_flow\_cost})}} v_{connection\_flow}(conn,n,d,s,t) \cdot p_{connection\_flow\_cost}(conn,s,t) \cdot p_{weight}(n,s,t) \cdot p_{duration}(t)$$

Start up costs

Start up costs associated with a specific unit can be included by defining the start_up_cost parameter. For all tuples of (unit, scenario, timestep) in the set `units_on_indices` for which this parameter is defined, a start up cost term is added to the objective function. The total start up costs can be expressed as:

$$v_{start\_up\_costs} = \sum_{\substack{(u,s,t) \in units\_on\_indices: \\ u \in ind(p_{start\_up\_cost})}} v_{units\_started\_up}(u,s,t) \cdot p_{start\_up\_cost}(u,s,t) \cdot p_{weight}(u,s,t)$$

Shut down costs

Shut down costs associated with a specific unit can be included by defining the shut_down_cost parameter. For all tuples of (unit, scenario, timestep) in the set `units_on_indices` for which this parameter is defined, a shut down cost term is added to the objective function. The total shut down costs can be expressed as:

$$v_{shut\_down\_costs} = \sum_{\substack{(u,s,t) \in units\_on\_indices: \\ u \in ind(p_{shut\_down\_cost})}} v_{units\_shut\_down}(u,s,t) \cdot p_{start\_up\_cost}(u,s,t) \cdot p_{weight}(u,s,t)$$

Ramping costs

To account for the ramping costs (up and down) associated with a specific unit, the parameters ramp_up_cost and ramp_down_cost can be defined. For all tuples of (unit, {node,node_group}, direction, scenario, timestep) in the sets `ramp_up_unit_flow_indices` and `ramp_down_unit_flow_indices` for which ramp_up_cost and ramp_down_cost are defined, respectively, a ramping cost term is added to the objective function. The total ramping costs can be expressed as:

$$
\begin{aligned}
&v_{ramp\_costs} \\
&= \sum_{\substack{(u,n,d,s,t) \in ramp\_up\_unit\_flow\_indices: \\ (u,n,d) \in ind(p_{ramp\_up\_cost})}} v_{ramp\_up\_unit\_flow}(u,n,d,s,t) \cdot p_{ramp\_up\_cost}(u,n,d,s,t) \cdot p_{weight}(n,s,t) \cdot p_{duration}(t) \\
&+ \sum_{\substack{(u,n,d,s,t) \in ramp\_down\_unit\_flow\_indices: \\ (u,n,d) \in ind(p_{ramp\_down\_cost})}} v_{ramp\_down\_unit\_flow}(u,n,d,s,t) \cdot p_{ramp\_down\_cost}(u,n,d,s,t) \cdot p_{weight}(n,s,t) \cdot p_{duration}(t)
\end{aligned}
$$

Reserve procurement costs

The procurement costs for reserves provided by a specific unit can be accounted for by defining the reserve_procurement_cost parameter. For all tuples of (unit, {node,node_group}, direction, scenario, timestep) in the set `unit_flow_indices` for which this parameter is defined, a reserve procurement cost term is added to the objective function. The total reserve procurement costs can be expressed as:

$$
\begin{aligned}
&v_{res\_proc\_costs} \\
&= \sum_{\substack{(u,n,d,s,t) \in unit\_flow\_indices: \\ (u,n,d) \in ind(p_{reserve\_procurement\_cost})}} v_{unit\_flow}(u,n,d,s,t) \cdot p_{reserve\_procurement\_cost}(u,n,d,s,t) \cdot p_{weight}(n,s,t) \cdot p_{duration}(t)
\end{aligned}
$$

Renewable curtailment costs

The curtailment costs of renewable units can be accounted for by defining the parameters curtailment_cost and unit_capacity. For all tuples of (unit, {node,node_group}, direction) for which these parameters are defined, and for which tuples (unit, scenario, timestep_long) exist in the set `units_on_indices`, and for which tuples (unit, {node,node_group}, direction, scenario, timestep_short) exist in the set `unit_flow_indices`, a renewable curtailment cost term is added to the objective function. The total renewable curtailment costs can be expressed as:

$$
\begin{aligned}
&v_{renewable\_curtailment\_costs} \\
&= \sum_{\substack{(u,n,d) \in ind(p_{unit\_capacity}): \\ u \in ind(p_{curtailment\_cost})}} \sum_{(u,s,t_{long}) \in units\_on\_indices} \sum_{(u,n,s,t_{short}) \in unit\_flow\_indices} \\
&\left(v_{units\_available}(u,s,t_{long}) \cdot p_{unit\_capacity}(u,n,d,s,t_{short}) \cdot p_{unit\_conv\_cap\_to\_flow}(u,n,d,s,t_{short}) \right. \\
&\left. - v_{unit\_flow}(u,n,d,s,t_{short})\right) \\
&\cdot p_{curtailment\_cost}(u,s,t_{short}) \cdot p_{weight}(n,s,t_{short}) \cdot p_{duration}(t_{short})
\end{aligned}
$$

Taxes

To account for taxes on certain commodity flows, the tax unit flow parameters (i.e., tax_net_unit_flow, tax_out_unit_flow and tax_in_unit_flow) can be defined. For all tuples of (unit, {node,node_group}, direction, scenario, timestep) in the set `unit_flow_indices` for which these parameters are defined, a tax term is added to the objective function. The total taxes can be expressed as:

$$
\begin{aligned}
&v_{taxes} \\
&= \sum_{\substack{(u,n,d,s,t) \in unit\_flow\_indices: \\ n \in ind(p_{tax\_net\_unit\_flow}) \& d = to\_node}} v_{unit\_flow}(u,n,d,s,t) \cdot p_{tax\_net\_unit\_flow}(n,s,t) \cdot p_{weight}(n,s,t) \cdot p_{duration}(t) \\
&- \sum_{\substack{(u,n,d,s,t) \in unit\_flow\_indices: \\ n \in ind(p_{tax\_net\_unit\_flow}) \& d = from\_node}} v_{unit\_flow}(u,n,d,s,t) \cdot p_{tax\_net\_unit\_flow}(n,s,t) \cdot p_{weight}(n,s,t) \cdot p_{duration}(t) \\
&+ \sum_{\substack{(u,n,d,s,t) \in unit\_flow\_indices: \\ n \in ind(p_{tax\_out\_unit\_flow}) \& d = from\_node}} v_{unit\_flow}(u,n,d,s,t) \cdot p_{tax\_out\_unit\_flow}(n,s,t) \cdot p_{weight}(n,s,t) \cdot p_{duration}(t) \\
&+ \sum_{\substack{(u,n,d,s,t) \in unit\_flow\_indices: \\ n \in ind(p_{tax\_in\_unit\_flow}) \& d = to\_node}} v_{unit\_flow}(u,n,d,s,t) \cdot p_{tax\_in\_unit\_flow}(n,s,t) \cdot p_{weight}(n,s,t) \cdot p_{duration}(t)
\end{aligned}
$$

Objective penalties

Penalty cost terms associated with the slack variables of a specific constraint can be accounted for by defining a node_slack_penalty parameter. For all tuples of ({node,node_group}, scenario, timestep) in the set `node_slack_indices` for which this parameter is defined, a penalty term is added to the objective function. The total objective penalties can be expressed as:

$$v_{objective\_penalties} = \sum_{(u,s,t) \in node\_slack\_indices} \left[v_{node\_slack\_neg}(n,s,t) - v_{node\_slack\_pos}(n,s,t) \right] \cdot p_{node\_slack\_penalty}(n,s,t) \cdot p_{weight}(n,s,t) \cdot p_{duration}(t)$$

« Constraints

Temporal Framework »

Temporal Framework

Spine Model aims to provide a high degree of flexibility in the temporal dimension across different components of the created model. This means that the user has some freedom to choose how the temporal aspects of different components of the model are defined. This freedom increases the variety of problems that can be tackled in Spine: from very coarse, long term models, to very detailed models with a more limited horizon, or a mix of both. The choice of the user on how this flexibility is used will lead to the temporal structure of the model.

The main components of flexibility consist of the following parts:

- The horizon that is modeled: end and start time
- Temporal resolution
- Possibility of a rolling optimization window
- Support for commonly used methods such as representative days

Part of the temporal flexibility in Spine is due to the fact that these options mentioned above can be implemented differently across different components of the model, which can be very useful when different markets are coupled in a single model. The resolution and horizon of the gas market can for example be taken differently than that of the electricity market. This documentation aims to give the reader insight in how these aspects are defined, and which objects are used for this.

We start by introducing the relevant objects with their parameters, and the relevant relationship classes for the temporal structure. Afterwards, we will discuss how this setting creates flexibility and will present some of the practical approaches to create a variety of temporal structures.

Objects, relationships, and their parameters

In this section, the objects and relationships will be discussed that form the temporal structure together.

Objects relevant for the temporal framework

For the objects, the relevant parameters will also be introduced, along with the type of values that are allowed, following the format below:

- 'parameter_name' : "Allowed value type"

model object

Each `model` object holds general information about the model at hand. Here we only discuss the time related parameters:

- model_start and model_end : "Date time value"

These two parameters define the model horizon. A Datetime value is to be taken for both parameters, in which case they directly mark respectively the beginning and end of the modeled time horizon.

- duration_unit (optional): "minute or hour"

This parameters gives the unit of duration that is used in the model calculations. The default value for this parameter is 'minute'. E.g. if the duration_unit is set to `hour`, a `Duration` of one `minute` gets converted into `1/60 hours` for the calculations.

- roll_forward (optional): "duration value"

This parameter defines how much the optimization window rolls forward in a rolling horizon optimization and should be expressed as a duration. In the practical approaches presented below, the rolling window optimization will be explained in more detail.

temporal_block object

A temporal block defines the properties of the optimization that is to be solved in the current window. Most importantly, it holds the necessary information about the resolution and horizon of the optimization.

- resolution (optional): "duration value" or "array of duration values"

This parameter specifies the resolution of the temporal block, or in other words: the length of the timesteps used in the optimization run.

- block_start (optional): "duration value" or "Date time value"

Indicates the start of this temporal block.

- block_end(optional): "duration value" or "Date time value"

Indicates the end of this temporal block.

Relationships relevant for the temporal framework

model__temporal_block relationship

In this relationship, a model instance is linked to a temporal block. If this relationship doesn't exist - the temporal block is disregarded from this optimization model.

model__default_temporal_block relationship

Defines the default temporal block used for model objects, which will be replaced when a specific relationship is defined for a model in `model__temporal_block`.

node__temporal_block relationship

This relationship will link a node to a temporal block.

units_on__temporal_block relationship

This relationship links the `units_on` variable of a unit to a temporal block and will therefore govern the time-resolution of the unit's online/offline status.

unit__investment_temporal_block relationship

This relationship sets the temporal dimensions for investment decisions of a certain unit. The separation between this relationship and the `units_on__temporal_block`, allows the user for example to give a much finer resolution to a unit's on- or offline status than to it's investment decisions.

model__default_investment_temporal_block relationship

Defines the default temporal block used for investment decisions, which will be replaced when a specific relationship is defined for a unit in `unit__investment_temporal_block`.

General principle of the temporal framework

The general principle of the Spine modeling temporal structure is that different temporal blocks can be defined and linked to different objects in a model. This leads to great flexibility in the temporal structure of the model as a whole. To illustrate this, we will discuss some of the possibilities that arise in this framework.

One single `temporal_block`

Single solve with single block

The simplest case is a single solve of the entire time horizon (so `roll_forward` not defined) with a fixed resolution. In this case, only one temporal block has to be defined with a fixed resolution. Each node has to be linked to this `temporal_block`.

Alternatively, a variable resolution can be defined by choosing an array of durations for the resolution parameter. The sum of the durations in the array then have to match the length of the temporal block. The example below illustrates an optimization that spans one day for which the resolution is hourly in the beginning and then gradually decreases to a 6h resolution at the end.

- `temporal_block_1`
  - `block_start`: 0h *(Alternative `DateTime`: e.g. 2030-01-01T00:00:00)*
  - `block_end`: 1D *(Alternative `DateTime`: e.g. 2030-01-02T00:00:00)*

- `resolution`: [1h 1h 1h 1h 2h 2h 2h 4h 4h 6h]

Note that, as mentioned above, the block_start and block_end parameters can also be entered as absolute values, i.e. `DateTime` values.

Rolling window optimization with single block

A model with a single `temporal_block` can also be optimized in a rolling horizon framework. In this case, the `roll_forward` parameter has to be defined in the `model` object. The `roll_forward` parameter will then determine how much the optimization moves forward with every step, while the size of the temporal block will determine how large a time frame is optimized in each step. To see this more clearly, let's take a look at an example.

Suppose we want to model a horizon of one week, with a rolling window size of one day. The `roll_forward` parameter will then be a duration value of 1d. If we take the `temporal_block` parameters `block_start` and `block_end` to be the duration values 0h and 1d respectively, the model will optimize each day of the week separately. However, we could also take the `block_end` parameter to be 2d. Now the model will start by optimizing day 1 and day 2 together, after which it keeps only the values obtained for the first day, and moves forward to optimize the second and third day together.

Again, a variable resolution can be implemented for the rolling window optimization. The sum of the durations must in this case match the size of the optimized window.

Advanced usage: multiple `temporal_block` objects

Single solve with multiple blocks

Disconnected time periods

Multiple temporal blocks can be used to optimize disconnected periods. Let's take a look at an example in which two temporal blocks are defined.

- `temporal_block_1`
  - `block_start`: 0h
  - `block_end`: 4h
- `temporal_block_2`
  - `block_start`: 12h
  - `block_end`: 16h

This example will lead to an optimization of the first four hours of the model horizon, and also of hour 12 to 16. By defining exactly the same relationships for the two temporal blocks, an optimization of disconnected periods is achieved for exactly the same model components. This leads to the possibility of implementing the widely used representative days method. If desired, it is possible to choose a different temporal resolution for the different `temporal_blocks`.

It is worth noting that dynamic variables like node_state and units_on merit special attention when using disconnected time periods. By default, when trying to access variables Variables outside the defined temporal_blocks, *SpineOpt.jl* assumes such variables exist but allows them to take any values within specified bounds. If fixed initial conditions for the disconnected periods are desired, one needs to use parameters such as fix_node_state or fix_units_on.

Different regions/commodities in different resolutions

Multiple temporal blocks can also be used to model different regions or different commodities with a different resolution. This is especially useful when there is a certain region or commodity of interest, while other elements are connected to this but require less detail. For this kind of usage, the relationships that are defined for the temporal blocks will be different, as shown in the example below.

- `temporal_blocks`
  - `temporal_block_1`
    - `resolution`: 1h
  - `temporal_block_2`
    - `resolution`: 2h
- `nodes`
  - `node_1`
  - `node_2`
- `node_temporal_block` relationships
  - `node_1_temporal_block_1`
  - `node_2_temporal_block_2`

Similarly, the on- and offline status of a unit can be modeled with a lower resolution than the actual output of that unit, by defining the `units_on_temporal_block` relationship with a different temporal block than the one used for the `node_temporal_block` relationship (of the node to which the unit is connected).

Rolling horizon with multiple blocks

Rolling horizon with different window sizes

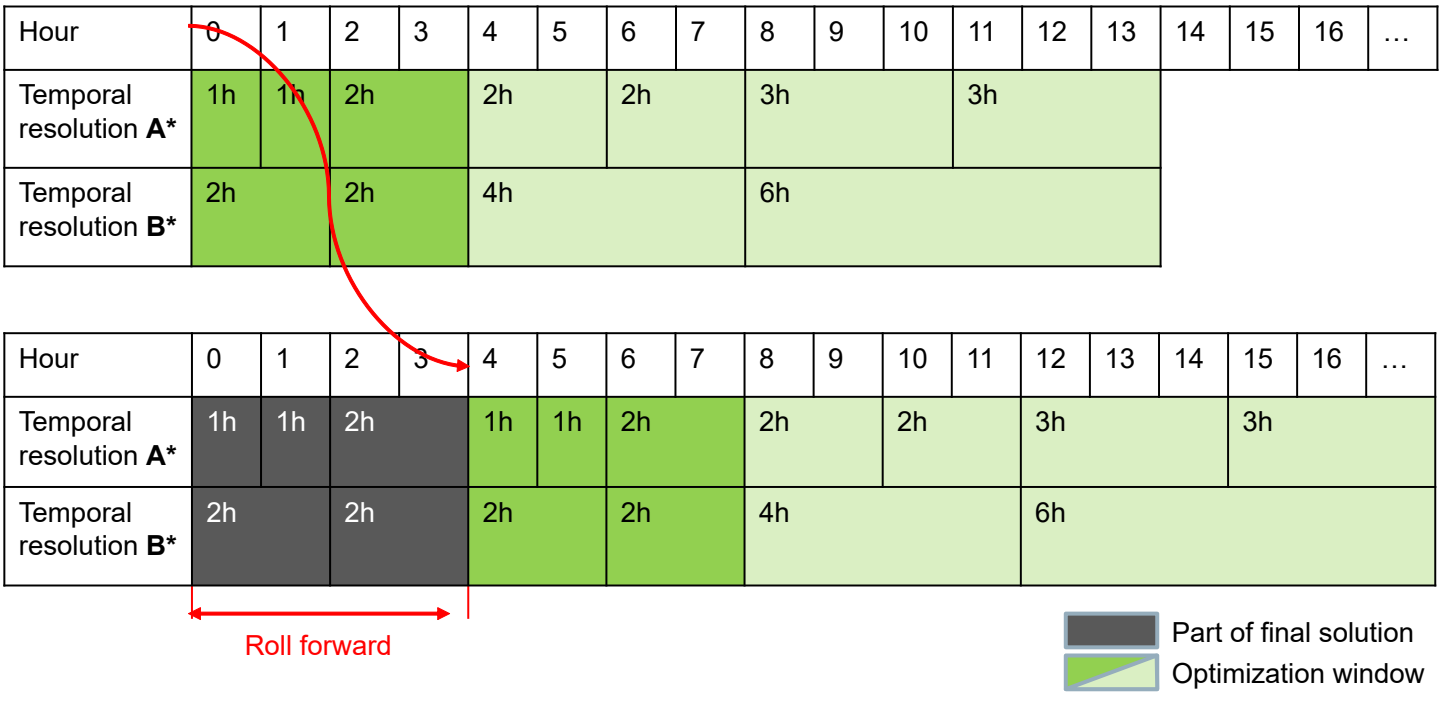
Similar to what has been discussed above in Different regions/commodities in different resolutions, different commodities or regions can be modeled with a different resolution in the rolling horizon setting. The way to do it is completely analogous. Furthermore, when using the rolling horizon framework, a different window size can be chosen for the different modeled components, by simply using a different `block_end` parameter. However, using different block_ends e.g. for interconnected regions should be treated with care, as the variables for each region will only be generated for their respective temporal_block, which in most cases will lead to inconsistent linking constraints.

Putting it all together: rolling horizon with variable resolution that differs for different model components

Below is an example of an advanced use case in which a rolling horizon optimization is used, and different model components are optimized with a different resolution. By choosing the relevant parameters in the following way:

- `model`
  - `roll_forward`: 4h
- `temporal_blocks`
  - `temporal_block_A`
    - `resolution`: [1h 1h 2h 2h 2h 3h 3h]
    - `block_end`: 14h
  - `temporal_block_B`
    - `resolution`: [2h 2h 4h 6h]
    - `block_end`: 14h
- `nodes`
  - `node_1`
  - `node_2`
- `node_temporal_block` relationships
  - `node_1_temporal_block_A`
  - `node_2_temporal_block_B`

The two model components that are considered have a different resolution, and their own resolution is also varying within the optimization window. Note that in this case the two optimization windows have the same size, but this is not strictly necessary. The image below visualizes the first two window optimizations of this model.



* A/B can correspond to different zones/regions (e.g., BE and FR) and/or a different market sector (e.g., Gas & Electricity)

« Objective

Stochastic Framework »

Stochastic Framework

Scenario-based stochastics in unit commitment and economic dispatch models typically only consider branching scenario trees. However, sometimes the available stochastic data doesn't span over the entire desired modelling horizon, or all the modelled phenomena. Especially with increasing interest in energy system integration and sector coupling, stochastic data of consistent quality and/or length might be hard to come by.

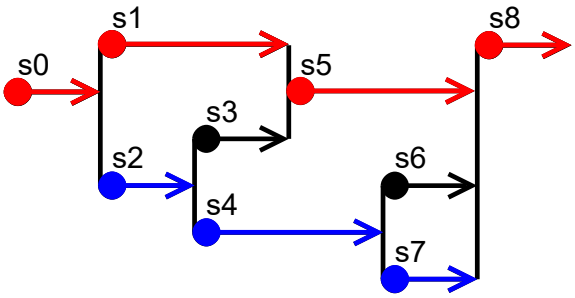
While these data issues can be circumvented by either cloning stochastic data across multiple scenario branches or generating dummy forecasts, they can result in inflated problem sizes. Furthermore, Ensuring realistic correlations between generated forecasts is extremely difficult, especially across multiple energy sectors.

The stochastic framework in *SpineOpt.jl* aims to support stochastic directed acyclic graphs (DAGs) instead of only branching trees, allowing for scenarios to converge later on in the modelled horizon. In addition, the framework allows for slightly different stochastic scenario graphs for different variables, making it easier to define e.g. variables common between all stochastic scenarios.

Key concepts

Here, we briefly describe the key concepts required to understand the stochastic framework:

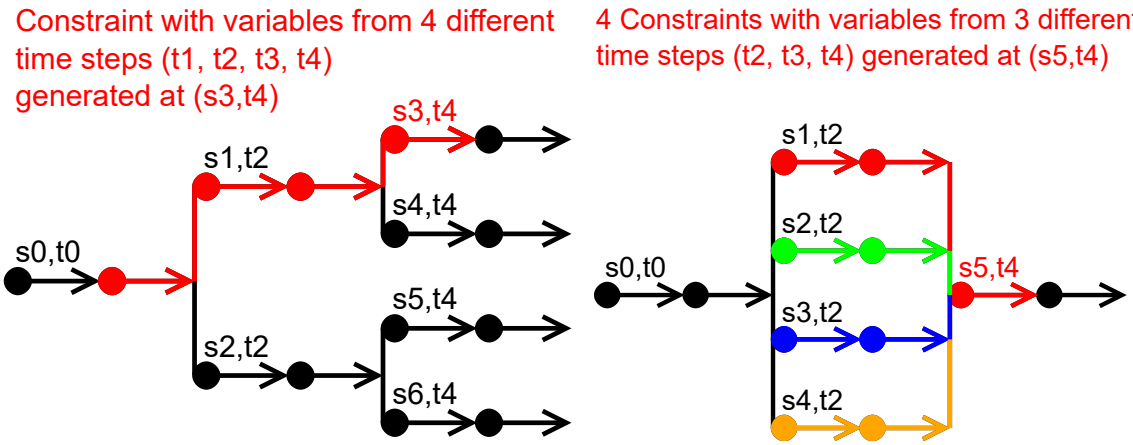
1. **stochastic_scenario** is essentially just a label for an alternative period of time, describing one possiblity of what may come to pass. Even in deterministic modelling with *SpineOpt.jl*, a single stochastic_scenario is required for labelling the deterministic timeline.
2. **Stochastic DAG** is the directed acyclic graph describing the parent_stochastic_scenario__child_stochastic_scenario relationships between the `stochastic scenarios`. The key difference between a *stochastic DAG* and a traditional *stochastic tree* is that the scenarios are allowed to have multiple parents, making it possible to converge scenarios into each other in addition to branching.
3. **Stochastic path** is a unique sequence of `stochastic scenarios` for traversing the *stochastic DAG*. Every *(finite) stochastic DAG* has a limited number of *full stochastic paths* that traverse it from roots *(scenarios without parents)* to leaves *(scenarios without children)*. Here, we use the term *stochastic path* to refer to any subset of scenarios within a *full stochastic path*.
4. **stochastic_structure** is essentially a "realization" of the *stochastic DAG*, with additional information like the stochastic_scenario_end and weight_relative_to_parents Parameters. These become relevant when we start discussing interactions between different `stochastic structures`.



The above figure presents an example *stochastic DAG* with the individual `stochastic scenarios` labelled from `s0-s8`. An example *full stochastic path* `[s0, s1, s5, s8]` is highlighted in red, while an example *stochastic path* `[s2, s4, s7]` is highlighted in blue.

General idea in brief

The major issue with *stochastic DAGs* compared to *stochastic trees*, is that indexing constraints that include variables from multiple time steps *(henceforth referred to as "dynamic constraints")* needs rethinking. With *stochastic trees*, constraints can always be unambiguously indexed using `(stochastic_scenario, last_time_step)`, since all `stochastic scenarios` only have a single parent. However, this is no longer the case for *stochastic DAGs*, as illustrated in the figures below:



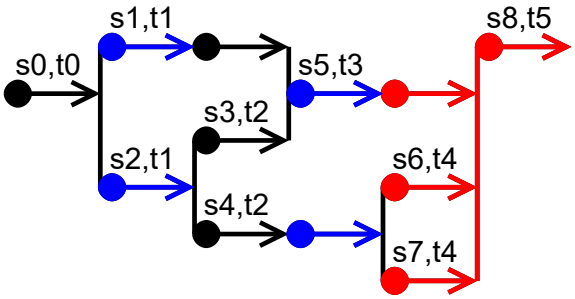
The example on the left illustrates the "traditional" indexing in branching *stochastic trees*, where backtracking through the tree always leads to unambiguous `(stochastic_scenario, time_step)` indices. The example on the right shows a similar situation in a *stochastic DAG*, where backtracking through the DAG leads to four different `(stochastic_scenario, time_step)` indices, and thus requires four constraints to be generated and indexed.

Stochastic path indexing

As discussed in the previous section, dynamic constraints in *stochastic DAGs* cannot be unambiguously indexed using a single `(stochastic_scenario, time_step)`. However, they *can* be unambiguously indexed using `(stochastic_path, time_step)`, where the *stochastic path* is the unique sequence of `stochastic scenarios` traversing the DAG. Since there are only a limited number of ways to traverse the DAG, represented by the *full stochastic paths*, we can identify the number of unique paths necessary for constraint generation as follows:

1. Identify all unique *full stochastic paths*, meaning all the possible ways of traversing the DAG from roots to leaves.
2. Find all the `stochastic scenarios` that are active on all the `time steps` included in the constraint.
3. Find all the unique *stochastic paths* by intersecting the set of active `stochastic scenarios` with the *full stochastic paths*.
4. Generate constraints over each unique *stochastic path* found in step 3.

Example dynamic constraint generation



The above figure shows examples of two different dynamic constraints generated in a *stochastic DAG*: the red constraint including variables from timesteps `t4-t5` and the blue constraint including variables from timesteps `t1, t3`. The *full stochastic paths* for traversing the above DAG are as follows:

1. `[s0, s1, s5, s8]`
2. `[s0, s2, s3, s5, s8]`
3. `[s0, s2, s4, s6, s8]`
4. `[s0, s2, s4, s7, s8]`

For the red constraint, the `stochastic scenarios` `s5-s8` are active on the `time steps` `t4-t5`. All the above *full stochastic paths* include at least two of the active `stochastic scenarios`, but full paths 1 and 2 both produce an identical path `[s5, s8]`, so the set of unique *stochastic paths* for the red constraint becomes:

1. `[s5, s8]`
2. `[s6, s8]`
3. `[s7, s8]`

There are no paths `[s5, s6], [s5, s7], [s6, s7]` since following the DAG one cannot start from `s5` and end up in `s6`, even though these `stochastic scenarios` are active.

The blue constraint illustrates a case where the time step range is non-continuous. The active `stochastic scenarios` on `t1, t3` are `s1, s2, s4, s5`, so again by comparing these to the *full stochastic paths* we get:

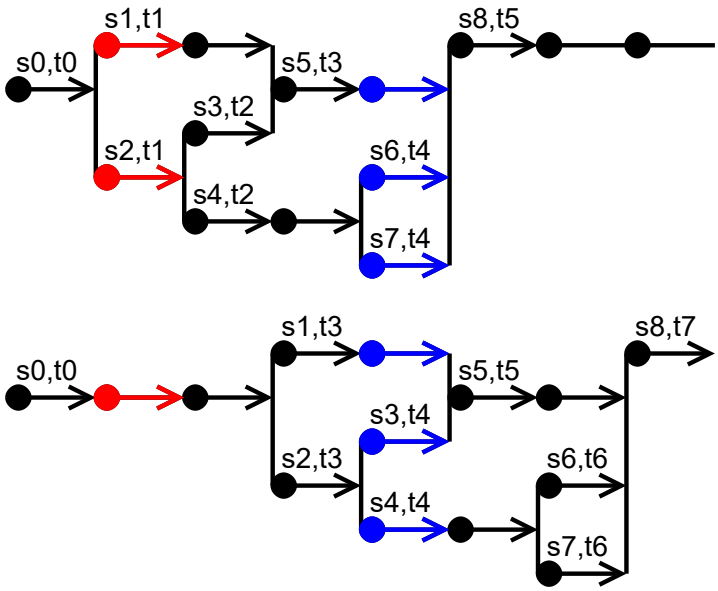
1. `[s1, s5]`
2. `[s2, s5]`

3. `[s2, s4]`

In this case, the *full stochastic paths* 3 and 4 both produce the path `[s2, s4]`, so only three unique constraints need to be generated. Again, the path `[s1, s4]` is invalid, since the DAG cannot be traversed from `s1` to `s4`.

Interaction between different stochastic structures

Stochastic path indexing in constraints also allows for "distorting" the *stochastic DAG* in different parts of the model. As long as the *stochastic DAG* itself isn't changed, meaning the parent_stochastic_scenario__child_stochastic_scenario relationships and the resulting *full stochastic paths*, we can actually define different `stochastic structures` and still be able to handle constraint generation between them. This is due to the fact that when determining the *stochastic paths*, it makes no difference whether we're looking at the same stochastic_structure at different `time steps`, or at two `stochastic structures`, one of which has been delayed, on the same `time step`. This is illustrated by the figure below:



The above represents constraint generation over two `stochastic structures`, where the lower stochastic_structure has been delayed in respect to the one above. Nevertheless, the procedure for finding the *stochastic paths* for the constraints remains identical to the previous example:

1. Identify all unique *full stochastic paths*, meaning all the possible ways of traversing the DAG. As long as the DAG remains the same between all the involved `stochastic structures`, the pathing remains the same.
2. Find all the `stochastic scenarios` that are active on all the `stochastic structures` and `time steps` included in the constraint.
3. Find all the unique stochastic paths by intersecting the set of active scenarios with the *full stochastic paths*.
4. Generate constraints over each unique stochastic path found in step 3.

Stochastics in the model data structure

While the Key concepts and General idea in brief sections go over the stochastic framework in *SpineOpt.jl* in a more general sense, here we'll go over how to set up stochastics using *SpineOpt.jl* data structure. Simple step-by-step examples are also provided in the Example of deterministic stochastics, Example of branching stochastics, and Example of converging stochastics sections further below. We won't go into too much detail about the related Object Classes, Relationship Classes, or Parameters, since those can be found in their respective sections. Introductions to these concepts can also be found in the Structural object classes and Structural relationship classes sections, if necessary.

Setting up the stochastic framework

As with all things in *SpineOpt.jl*, you'll want to start with adding the desired number of objects to the relevant Object Classes, as one cannot define relationships over objects that don't exist. For the stochastic framework, this means creating at least one stochastic_scenario and stochastic_structure object each. This needs to be done even if your model is fully deterministic, as even the deterministic structure needs to be labelled for *SpineOpt.jl* to recognize that it exists.

Next, if your model has multiple stochastic_scenario objects, you'll want to define how they are related using the parent_stochastic_scenario__child_stochastic_scenario relationship. This relationship essentially defines the *stochastic DAG*, as well as all the possible *stochastic paths*, explained in the Key concepts section. Unless the parent_stochastic_scenario__child_stochastic_scenario relationship is defined, there won't be a *stochastic DAG*, and all stochastic_scenario objects will be assumed to be completely independent of each other.

Now that you've set up the desired stochastic_scenario and stochastic_structure objects, as well as defined the *stochastic DAG* using the parent_stochastic_scenario__child_stochastic_scenario relationship, it's time to define the properties of the stochastic_structure objects using the stochastic_structure__stochastic_scenario relationship, and the stochastic_scenario_end and weight_relative_to_parents Parameters therein. You'll always have to define at least one stochastic_structure__stochastic_scenario relationship, as the stochastic_structure object is what connects the Systemic object classes to the stochastic framework.
stochastic_structure__stochastic_scenario relationship holds two key Parameters:

- **weight_relative_to_parents** defines the coefficient the corresponding stochastic_scenario has in the Objective function, and needs to be defined for each stochastic_scenario included in the stochastic_structure. The weight is relative to the parents of the [stochastic_scenario], and is calculated as presented below.

```
# For root `stochastic_scenarios` (meaning no parents)

weight(scenario) = weight_relative_to_parents(scenario)

# If not a root `stochastic_scenario`
```

```
weight(scenario) = sum([weight(parent) * weight_relative_to_parents(scenario)] for p
```

- **stochastic_scenario_end** is a `Duration` type parameter that tells when the stochastic_scenario ends in relation to the start of the current optimization. When defined, the stochastic_scenario ends at the defined point in time, and spawns its children according to parent_stochastic_scenario__child_stochastic_scenario, if any. **Note that this means the children are included in the stochastic_structure, even without an explicit relationship!** If stochastic_scenario_end isn't defined, the stochastic_scenario is assumed to go on indefinetely.

Finally, with all the pieces in place, we'll need to connect the defined stochastic_structure objects to the desired objects in the Systemic object classes using the Structural relationship classes like node__stochastic_structure etc. Here, we essentially tell which parts of the modelled system use which stochastic_structure. Since creating each of these relationships individually can be a bit of a pain, there are a few Meta relationship classes like the model__default_stochastic_structure, that can be used to set model-wide defaults that are used if specific relationships are missing.

Example of deterministic stochastics

Here, we'll demonstrate step-by-step how to create the simplest possible stochastic frame: the fully deterministic one.

1. Create a stochastic_scenario called e.g. `realization` and a stochastic_structure called e.g. `deterministic`.
2. We can skip the parent_stochastic_scenario__child_stochastic_scenario relationship, since there isn't a *stochastic DAG* in this example, and the default behaviour of each stochastic_scenario being independent works for our purposes *(only one stochastic_scenario anyhow)*.
3. Create the stochastic_structure__stochastic_scenario relationship for `(deterministic, realization)`, and set its weight_relative_to_parents parameter to 1. We don't need to define the stochastic_scenario_end parameter, as we want the `realization` to go on indefinitely.
4. Relate the `deterministic` stochastic_structure to all the desired system objects using the appropriate Structural relationship classes, or use the model-level default Meta relationship classes.

Example of branching stochastics

Here, we'll demonstrate step-by-step how to create a simple branching stochastic tree, where one scenario branches into three at a specific point in time.

1. Create four stochastic_scenario objects called e.g. `realization`, `forecast1`, `forecast2`, and `forecast3`, and a stochastic_structure called e.g. `branching`.
2. Define the *stochastic DAG* by creating the parent_stochastic_scenario__child_stochastic_scenario relationships for `(realization, forecast1)`, `(realization, forecast2)`, and `(realization,`

`forecast3)`.

3. Create the stochastic_structure__stochastic_scenario relationship for `(branching, realization)`, `(branching, forecast1)`, `(branching, forecast2)`, and `(branching, forecast3)`.
4. Set the weight_relative_to_parents parameter to 1 and the stochastic_scenario_end parameter e.g. to `6h` for the stochastic_structure__stochastic_scenario relationship `(branching, realization)`. Now, the `realization` stochastic_scenario will end after 6 hours of `time steps`, and its children *(`forecast1`, `forecast2`, and `forecast3`)* will become active.
5. Set the weight_relative_to_parents Parameters for the `(branching, forecast1)`, `(branching, forecast2)`, and `(branching, forecast3)` stochastic_structure__stochastic_scenario relationships to whatever you desire, e.g. 0.33 for equal probabilities across all forecasts.
6. Relate the `branching` stochastic_structure to all the desired system objects using the appropriate Structural relationship classes, or use the model-level default Meta relationship classes.

Example of converging stochastics

Here, we'll demonstrate step-by-step how to create a simple *stochastic DAG*, where both branching and converging occurs. This example relies on the previous Example of branching stochastics, but adds another stochastic_scenario at the end, which is a child of the `forecast1`, `forecast2`, and `forecast3` scenarios.

1. Follow the steps 1-5 in the previous Example of branching stochastics, except call the stochastic_structure something different, e.g. `converging`.
2. Create a new stochastic_scenario called e.g. `converged_forecast`.
3. Alter the *stochastic DAG* by creating the parent_stochastic_scenario__child_stochastic_scenario relationships for `(forecast1, converged_forecast)`, `(forecast2, converged_forecast)`, and `(forecast3, converged_forecast)`. Now all three forecasts will converge into a single `converged_forecast`.
4. Add the stochastic_structure__stochastic_scenario relationship for `(converging, converged_forecast)`, and set its weight_relative_to_parents parameter to 1. Now, all the probability mass in `forecast1`, `forecast2`, and `forecast3` will be summed up back to the `converged_forecast`.
5. Set the stochastic_scenario_end Parameters of the stochastic_structure__stochastic_scenario relationships `(converging, forecast1)`, `(converging, forecast2)`, and `(converging, forecast3)` to e.g. `12h`, so that all three scenarios end at the same time and the `converged_forecast` becomes active.
6. Relate the `converging` stochastic_structure to all the desired system objects using the appropriate Structural relationship classes, or use the model-level default Meta relationship classes.

Working with stochastic updating data

Now that we've discussed how to set up stochastics for SpineOpt, let's focus on stochastic data. The most complex form of input data SpineOpt can currently handle is both *stochastic* and *updating*, meaning that the values the parameter takes can depend on both the stochastic_scenario, and the *analysis time (first time step)* of each solve. However, just *stochastic* or just *updating* cases are supported as well, using the same input data format.

In SpineOpt, stochastic data uses the `Map` data type from *SpineInterface.jl*. Essentially, `Maps` are general indexed data containers, which SpineOpt tries to interpret as stochastic data. Every time SpineOpt calls a parameter, it passes the stochastic_scenario and *analysis time* as keyword arguments to the parameter, but depending on the parameter type, it doesn't necessarily do anything with that information. For `Map` type parameters, those keyword arguments are used for navigating the indices of the `Map` to try and find the corresponding value. If the `Map` doesn't include the stochastic_scenario index it's looking for, it assumes there's no *stochastic* information in the `Map` and carries on to search for *analysis time* indices. This logic is useful for defining both *stochastic* and *updating* data, as well as either case by itself, as shown in the following examples.

Example of stochastic data

By *stochastic data*, we mean parameter values that depend only on the stochastic_scenario. In such a case, the input data must be formatted as a `Map` with the following structure

stochastic_scenario	value
`scenario1`	`value1`
`scenario2`	`value2`

where stochastic_scenario indices are simply `Strings` corresponding to the names of the stochastic_scenario objects. The *values* can be whatever data types *SpineInterface.jl* supports, like `Constants`, `DateTimes`, `Durations`, or `TimeSeries`. In the above example, the parameter will take `value1` in `scenario1`, and `value2` in `scenario2`. Note that since there's no *analysis time* index in this example, the *values* are used regardless of the *analysis time*.

Example of updating data

By *updating data*, we mean parameter values that depend only on the *analysis time*. In such a case, the input data must be formatted as a `Map` with the following structure

analysis time	value
`2000-01-01T00:00:00`	`value1`
`2000-01-01T12:00:00`	`value2`

where the *analysis time* indices are `DateTime` values. The *values* can be whatever data types *SpineInterface.jl* supports, like `Constants`, `DateTimes`, `Durations`, or `TimeSeries`. In the above example, the parameter will take `value1` if the first time step of the current simulation is between `2000-01-01T00:00:00` and `2000-01-01T12:00:00`, and `value2` if the first time step of the simulation is after `2000-01-01T12:00:00`. Note that since there's no stochastic_scenario index in this example, the *values* are used regardless of the stochastic_scenario.

Example of stochastic updating data

By *stochastic updating data*, we mean parameter values that depend on both the stochastic_scenario and the *analysis time*. In such a case, the input data must be formatted as a `Map` with the following structure

stochastic_scenario	analysis time	value
`scenario1`	`2000-01-01T00:00:00`	`value1`
`scenario1`	`2000-01-01T12:00:00`	`value2`
`scenario2`	`2000-01-01T00:00:00`	`value3`
`scenario2`	`2000-01-01T12:00:00`	`value4`

where the stochastic_scenario indices are simply `Strings` corresponding to the names of the stochastic_scenario objects, and the *analysis time* indices are `DateTime` values. The *values* can be whatever data types *SpineInterface.jl* supports, like `Constants`, `DateTimes`, `Durations`, or `TimeSeries`. In the above example, the parameter will take `value1` if the first time step of the current simulation is between `2000-01-01T00:00:00` and `2000-01-01T12:00:00` and the parameter is called in `scenario1`, and `value3` in `scenario2`. If the first time step of the current simulation is after `2000-01-01T12:00:00`, the parameter will take `value2` in `scenario1`, and `value4` in `scenario2`.

Constraint generation with stochastic path indexing

Every time a constraint might refer to variables either on different time steps or on different `stochastic scenarios` (meaning different `nodes` or `units`), the constraint needs to use stochastic path indexing in order to be correctly generated for arbitrary stochastic DAGs. In practise, this means following the procedure outlined below:

1. Identify all unique *full stochastic paths*, meaning all the possible ways of traversing the DAG. This is done along with generating the stochastic structure, so no real impact on constraint generation.
2. **Find all the `stochastic scenarios` that are active on all the `stochastic structures` and `time slices` included in the constraint.**
3. **Find all the unique stochastic paths by intersecting the set of active scenarios with the *full stochastic paths*.**

4. Generate constraints over each unique stochastic path found in step 3.

Steps 2 and 3 are the crucial ones, and are currently handled by separate `constraint_<constraint_name>_indices` functions. Essentially, these functions go through all the variables on all the time steps included in the constraint, collect the set of active `stochastic_scenarios` on each time step, and then determine the unique active stochastic paths on each time step. The functions pre-form the index set over which the constraint is then generated in the `add_constraint_<constraint_name>` functions.

« Temporal Framework

Unit Commitment »

Unit commitment

To incorporate technical detail about (clustered) unit-commitment statuses of units, the online, started and shutdown status of units can be tracked and constrained in SpineOpt. In the following, relevant relationships and parameters are introduced and the general working principle is described.

Key concepts for unit commitment

Here, we briefly describe the key concepts involved in the representation of (clustered) unit commitment models:

- units_on is an optimization variable that holds information about the on- or offline status of a unit. Unit commitment restrictions will govern how this variable can change through time.
- units_on__temporal_block is a relationship linking the `units_on` variable of this unit to a specific temporal_block object. The temporal block holds information on the temporal scope and resolution for which the variable should be optimized.
- online_variable_type is a method parameter and can take the values `unit_online_variable_type_binary`, `unit_online_variable_type_integer`, `unit_online_variable_type_linear`. If the binary value is chosen, the units status is modelled as a binary (classic UC). For clustered unit commitment units, the integer type is applicable. Note that if the parameter is not defined, the default will be linear. If the units status is not crucial, this can reduce the computational burden.
- number_of_units defines how many units of a certain unit type are available. Typically this parameter takes a binary (UC) or integer (clustered UC) value. To avoid confusion the following distinction will be made in this document: `unit` will be used to identify a Spine unit object, which can have multiple `members`. Together with the `unit_availability_factor`, this will determine the maximum number of members that can be online at any given time. (Thus restricting the `units_on` variable). The default value for this parameter is 1. It is possible to allow the model to increase the `number_of_units` itself, through Investment Optimization
- unit_availability_factor: (number value or time series). Is the fraction of the time that this unit is considered to be available, by acting as a multiplier on the capacity. A time series can be used to indicate the intermittent character of renewable generation technologies.
- min_up_time: (duration value). Sets the minimum time that a unit has to stay online after a startup. Inclusion of this parameter will trigger the creation of the constraint on Minimum up time (basic version)

- min_down_time: (duration value). Sets the minimum time that a unit has to stay offline after a shutdown. Inclusion of this parameter will trigger the creation of the constraint on Minimum down time (basic version)
- minimum_operating_point: (number value) limits the minimum value of the `unit_flow` variable for a unit which is currently online. Inclusion of this parameter will trigger the creation of the Constraint on minimum operating point
- start_up_cost: "number value". Cost associated with starting up a unit.
- shut_down_cost: "number value". Cost associated with shutting down a unit.

Illustrative unit commitment examples

Step 1: defining the number of members of a unit type

A spine unit can represent multiple members. This can be incorporated in a model by setting the number_of_units parameter to a specific value. For example, if we define a single unit in a model as follows:

- `unit_1`
  - `number_of_units`: 2

And we link the unit to a certain `node_1` with a unit__to_node relationship.

- `unit_1_to__node_1`

The single spine unit defined here, now represents two members. This means that a single unit_flow variable will be created for this unit, but the restrictions as imposed by the Ramping and Reserves framework will be adapted to reflect the fact that there are two members present, thus doubling the total capacity.

Step 2: choosing the online_variable_type

Next, we have to decide the online_variable_type for this unit, which will restrict the kind of values that the units_on variable can take. This basically comes down to deciding if we are working in a classical UC framework (`unit_online_variable_type_binary`), a clustered UC framework (`unit_online_variable_type_integer`), or a relaxed clustered UC framework (`unit_online_variable_type_linear`), in which a non-integer number of units can be online.

The classical UC framework can only be applied when the `number_of_units` equals 1.

Step 3: imposing a minimum operating point

The output of an online unit to a specific node can be restricted to be above a certain minimum by choosing a value for the minimum_operating_point parameter. This parameter is defined for the unit__to_node relationship, and is given as a fraction of the unit_capacity. If we continue with the example above, and define the following objects, relationships, and parameters:

- `unit_1`
  - `number_of_units`: 2
  - `unit_online_variable_type`: "unit_online_variable_type_integer"
- `unit_1_to__node_1`
  - `minimum_operating_point`: 0.2
  - `unit_capacity`: 200

It can be seen that in this case the unit_flow form `unit_1` to `node_1` must for any timestep t be larger than $units\_on(t) * 0.2 * 200$

Step 4: imposing a minimum up or down time

Spine units can also be restricted in their commitment status with minimum up- or down times by choosing a value for the min_up_time or min_down_time respectively. These parameters are defined for the unit object, and should be duration values. We can continue the example and add a minimum up time for the unit:

- `unit_1`
  - `number_of_units`: 2
  - `unit_online_variable_type`: "unit_online_variable_type_integer"
  - `min_up_time`: 2h
- `unit_1_to__node_1`
  - `minimum_operating_point`: 0.2
  - `unit_capacity`: 200

Whereas the `units_on` variable was restricted (before inclusion of the `min_up_time` parameter) to be smaller than or equal to the `number_of_units` for any timestep t, it now has to be smaller than or equal to the `number_of_units` decremented with the units_started_up summed over the timesteps that include `t - min_up_time`. This implies that a unit which has started up, has to stay online for at least the `min_up_time`

To consider a simple example let's assume that we have a model with a resolution of 1h. Suppose that before `t`, there is no member of the unit online and in timestep `t -> t + 1h`, one member starts up. Another member starts up in timestep `t + 1h \-> t + 2h`. The first startup, along with the minimum up time of 2 hours implies that the `units_on` variable of this unit has now changed to 1 in timestep `t -> t + 1h` and can not go back to 0 in timestep `t-> t + 1h -> t + 2h`. The second startup further

restricts the number of units that are allowed to be online, it can be seen that the following restrictions apply when both startups are combined with the minimum up time of 2h:

- `t-> t + 1h` : $units\_on = 1$
- `t + 1h -> t + 2h`: $units\_on = 2$
- `t + 2h-> t + 3h`: $units\_on \in 1, 2$
- `t + 3h-> t + 4h`: $units\_on \in 0, 1, 2$

The minimum down time restrictions operate in very much the same way, they simply impose that units that have been shut down, have to stay offline for the chosen period of time.

Step 5: allocationg a cost to startups or shutdowns

Costs can be allocated to startups or shutdowns by choosing a value for the start_up_cost or shut_down_cost respectively.

Step 6: defining unit availabilities

By defining a unit_availability_factor, the fact that typical members are not available all the time can be reflected in the model.

Typically, units are not available 100% of the time, due to scheduled maintenance, unforeseen outages, or other things. This can be incorporated in the model by setting the `unit_availability_factor` to a fractional value. For each timestep in the model, an upper bound is then imposed on the `units_on` variable, equal to `number_of_units` $*$ `unit_availability_factor`. This parameter can not be used when the `online_variable_type` is binary. It should also be noted that when the `online_variable_type` is of integer type, the aforementioned product must be integer as well, since it will determine the value of the `units_available` parameter which is restricted to integer values. The default value for this parameter is 1.

The `unit_availability_factor` can also be taken as a timeseries. By allowing a different availability factor for each timestep in the model, it can perfectly be used to represent intermittent technologies of which the output cannot be fully controlled.

« Stochastic Framework

Ramping and Reserves »

Ramping and Reserves

To enable the representation of units with a high level of technical detail, the ramping ability of units can be constrained in SpineOpt. This means that the user has the freedom to impose restrictions on the change in output of units between consecutive timesteps, for online (spinning) units, units starting up and units shutting down. In this section, the concept of ramps in SpineOpt will be introduced. Furthermore, the use of reserves will be explained.

Relevant objects, relationships and parameters

Everything that is related to ramping is defined in parameters of either the unit__to_node, unit__from_node, or unit__to_node_group relationship. Generally speaking, the ramping constraints will impose restrictions on the change in the unit_flow variable between two consecutive timesteps.

All parameters that limit the ramping abilities of a unit are expressed as a fraction of the unit capacity. This means that a value of 1 indicates the full capacity of a unit.

The discussion here will be kept conceptual, for the mathematical formulation the reader is referred to the Ramping and reserve constraints

Constraining spinning ramps

- unit_capacity: limit the maximum value of the `unit_flow` variable for a unit which is currently online. Inclusion of this parameter will trigger the creation of the Define unit/technology capacity constraint.
- ramp_up_limit : limit the maximum increase in the `unit_flow` variable between two consecutive timesteps for which the unit is online. The parameter is given as a fraction of the unit_capacity parameter. Inclusion of this parameter will trigger the creation of the Constraint on spinning upwards ramp_up
- ramp_down_limit : limit the maximum decrease in the `unit_flow` variable between two consecutive timesteps for which the unit is online. The parameter is given as a fraction of the unit_capacity parameter. Inclusion of this parameter will trigger the creation of the Constraint on spinning downward ramps
- ramp_up_cost : cost associated with upward ramping
- ramp_down_cost : cost associated with downward ramping

Constraining shutdown ramps

- max_shutdown_ramp : limit the maximum of the `unit_flow` variable for the timestep right before a shutdown. The parameter is given as a fraction of the unit_capacity parameter. Inclusion of this parameter will trigger the creation of the Constraint on maximum downward shut down ramps
- min_shutdown_ramp : limit the minimum of the `unit_flow` variable for the timestep right before a shutdown. The parameter is given as a fraction of the unit_capacity parameter. Inclusion of this parameter will trigger the creation of the Constraint on minimum downward shut down ramps

Constraining startup ramps

- max_startup_ramp : limit the maximum of the `unit_flow` variable for the timestep right after a start-up. The parameter is given as a fraction of the unit_capacity parameter. Inclusion of this parameter will trigger the creation of the Constraint on maximum upward start up ramp_up
- min_startup_ramp : limit the minimum of the `unit_flow` variable for the timestep right after a start-up. The parameter is given as a fraction of the unit_capacity parameter. Inclusion of this parameter will trigger the creation of the Constraint on minimum upward start up ramp_up

General principle and example use cases

The general principle of the Spine modelling ramping constraints is that all of these parameters can be defined separately for each unit. This allows the user to incorporate different units (which can either represent a single unit or a technology type) with different flexibility characteristics.

It should be noted that it is perfectly possible to omit all of the constraining parameters mentioned above. However, once either of the ramping parameters is defined, it is necessary to also assign values to the other parameters. E.g. if a user only wants to restrict the spinning ramp up capability of a unit, one also has to assign values to the `max_startup_ramp`, `min_Shutdown_Ramp` etc.

Illustrative examples

Step 1: Simple case of unrestricted unit

When none of the ramping parameters mentioned above are defined, the unit is considered to have full ramping flexibility. This means that in any given timestep, its output can be any value between 0 and its capacity, regardless of what the output of the unit was in the previous timestep, and regardless of the on- or offline status or the unit in the previous timestep. Provided that this does not conflict with the Unit commitment restrictions that are defined for this unit. Parameter values for a `unit__node` relationship are illustratively given below.

- `max_shutdown_ramp` : 1
- `min_shutdown_ramp` : 0
- `max_start_up_ramp` : 1
- `min_start_up_ramp` : 0
- `ramp_up_limit` : 1

- `ramp_down_limit` : 1
- `unit_capacity` : 200

Step 2: Spinning ramp restriction

A unit which is only restricted in spinning ramping can be created by changing the `ramp_up/down_limit` parameters:

- `ramp_up_limit` : 0.2
- `ramp_down_limit` : 0.4

This parameter choice implies that the unit's output between two consecutive timesteps can change with no more than $0.2 * 200$ and no less than $0.4 * 200$. For example, when the unit is running at an output of 100 in some timestep t, its output for the next timestep must be somewhere in the interval $[20, 140]$. Unless it shuts down completely.

Step 3: Shutdown restrictions

By changing the parameter `max_shutdown_ramp` in the previous example, an additional restriction is imposed on the maximum output of the unit from which it can go offline.

- `max_shutdown_ramp` : 0.5
- `min_shutdown_ramp` : 0.3

When this unit goes offline in a given timestep $t + 1$, the output of the unit must be below $0.5 * 200 = 100$ in the timestep t before that. Similarly, the parameter `min_shutdown_ramp` can be used to impose a minimum output value in the timestep before a shutdown. For example, a value of 0.3 in this example would mean that the unit can not be running below an output of 60 in timestep t.

Step 4: Startup restrictions

The startup restrictions are very similar to the shutdown restrictions, but of course apply to units that are starting up. Consider for example the same unit as in the example above, but now with a `max_start_up_ramp` equal to 0.4 and `min_start_up_ramp` equal to 0.2:

- `max_start_up_ramp` : 0.4
- `min_start_up_ramp` : 0.2

When the unit is offline in timestep t and comes online in timestep $t + 1$, its output in timestep $t + 1$ will be restricted to the interval $[40, 80]$.

Reserve concept

To include a requirement of reserve provision in a model, SpineOpt offers the possibility of creating reserve nodes. Of course reserve provision is different from regular operation, because the reserved

capacity does not actually get activated. In this section, we will take a look at the things that are particular for a reserve node.

Defining a reserve node

To define a reserve node, the following parameters have to be defined for the relevant node:

- is_reserve_node : this boolean parameter indicates that this node is a reserve node.
- upward_reserve : this boolean parameter indicates that the demand for reserve provision of this node concerns upward reserves.
- downward_reserve : this boolean parameter indicates that the demand for reserve provision of this node concerns downward reserves.
- reserve_procurement_cost: (optional) this parameter indicates the procurement cost of a unit for a certain reserve product and can be define on a unit__to_node or unit__from_node relationship.

Defining a node group

SpineOpt allows the user to constrain ramping abilities of units that are linked to multiple nodes by defining node groups. This is especially relevant for reserve provision because a unit that provides reserves is linked to a regular, as well as a reserve node. It is then possible to constrain the unit's ramping for the combination of regular operation and reserve provision.

Since reserve provision in fact literally reserves part of the capacity of a unit, the demand of the reserve node will be subtracted from the part that is available for regular operation. The section below will discuss how this works in SpineOpt by means of an example.

Since the demand of the nodes is defined on the individual node level (and the node group has no demand), the balance type of the group node should be set to `balance_type_none`.

Ramping constraints on a node group with one reserve node

Reserves step 1: simple case of unrestricted unit

Let's assume that we have one unit and two nodes in a model, one for reserves and one for regular demand. The unit is then linked by the `unit__to_node` relationships to each node individually, and on top of that, it is linked to a node group containing both nodes.

The ramping of the unit can now be constrained by defining the same parameters as before, but now for the node group. As before, the simplest case is a unit that is only restricted by its capacity:

- `max_shutdown_ramp` : 1

- `min_shutdown_ramp` : 0
- `max_start_up_ramp` : 1
- `min_start_up_ramp` : 0
- `ramp_up_limit` : 1
- `ramp_down_limit` : 1
- `unit_capacity` : 200

The capacity restriction now implies that the sum of the reserve demand and regular demand cannot exceed the capacity of the unit. For example: when the reserve node has a demand of `10` in timestep `t`, the `unit_flow` variable to the regular node must be smaller than or equal to `190`.

Reserves step 2: Spinning ramp restriction

The unit can be restricted only in spinning ramping, as in the previous example, by defining the `ramp_up/down_limit` parameters in the `unit__to_node` relationship **for the node group**:

- `ramp_up_limit` : **0.2**
- `ramp_down_limit` : **0.4**

This parameter choice implies that the unit's flow to the regular demand node between two consecutive timesteps can change with no more than $0.2 * 200 - upward\_reserve\_demand$ and no less than $0.4 * 200 - downward\_reserve\_demand$. For example, when the unit is running at an output of 100 in some timestep t, and there is an upward reserve demand of `10` its output for the next timestep must be somewhere in the interval $[20, 130]$.

It can be seen in this example that the demand for reserves is subtracted from both the generation capacity, and the ramping capacity of the unit that is available for regular operation. This stems from the fact that in providing reserve capacity, the unit is expected to be able to provide the demanded reserve within one timestep.

Reserves Step 3: Non-spinning reserves

Units can also be allowed to provide non-spinning reserves, through shutdowns and startups. This can be done by using the following parameters in the `unit__to_node` relationship **for the reserve node** :

- `max_res_startup_ramp`
- `min_res_startup_ramp`
- `max_res_shutdown_ramp`
- `min_res_shutdown_ramp`
- `unit_capacity`

These parameters are constraining reserve provision in exactly the same way as their equivalents for regular operation. Note that it is now necessary to define a capacity of the unit with respect to the

reserve node. The ramping parameters will then be interpreted as fractions of this specific capacity. The unit's overall capacity can be different than its capacity for reserve provision.

A unit which can provide both spinning and non-spinning reserves can be defined as follows:

Parameters to be defined for unit to node group relationship

- `max_shutdown_ramp` : 1
- `min_shutdown_ramp` : 0
- `max_start_up_ramp` : 1
- `min_start_up_ramp` : 0
- `ramp_up_limit` : 0.2
- `ramp_down_limit` : 0.4
- `unit_capacity` : 200

Parameters to be defined for unit to reserve node relationship

- `max_res_startup_ramp`: 0.5
- `min_res_startup_ramp`: 0.1
- `unit_capacity`: 150

The spinning reserve and ramping restrictions now remain the same as above, but on top of that the unit is able to provide non-spinning upward reserves when it is offline. In this particular example, the contribution of the offline unit to upward reserves can be anything in the interval `[15,75]`.

Using node_groups for both combined and individual restrictions

It can be seen from the example above that when a node group is defined, ramping restrictions can be imposed both on the group level (thus for the unit as a whole) as well as for the individual nodes. If, for example a `ramp-up-limit` is defined for the node group, the sum of upward ramping of the two nodes will be restricted by this parameter, but it is still possible to limit the individual flows to the nodes as well. We will now discuss an example of this for the `ramp_up_limit`, but this also holds for other parameters.

Let's continue with the example above, where an online unit is capable of ramping up by 20% of its capacity and down by 40%. We might want to impose tighter restrictions for upward reserve provision than the ramping in overall operation (e.g. because the reserved capacity has to be available in a shorter time than the duration_unit). One can then simply define an additional parameter for the unit to reserve node relationship as follows.

- `ramp_up_limit` : 0.15

Which now restricts the spinning upward ramping provision of the unit to 15% of its capacity, as defined for the reserve node. In this case, the change in the unit's flow to the regular demand node between two consecutive timesteps is still limited to the interval $[0.2 * 200 - upward\_reserve\_demand, 0.4 * 200 - downward\_reserve\_demand]$. But the upward reserves that it can provide has an upper bound of `$150 * 0.15$.

« Unit Commitment

Investment Optimization »

Investment Optimization

SpineOpt offers numerous ways to optimise investment decisions energy system models and in particular, offers a number of methologogies for capturing increased detail in investment models while containing the impact on run time. The basic principles of investments will be discussed first and this will be followed by more advanced approaches.

Key concepts for investments

Investment Decisions These are the investment decisions that SpineOpt currently supports. At a high level, this means that the activity of the entities in question is controlled by an investment decision variable. The current implementation supports investments in :

- unit:
- connection
- **Storage** - Note: while the above investment decisions correspond to an object class (i.e.) an investment in a unit or a connection, **Storages** are not an object class in themselves and are rather a property of a node. As such, a storage investment controls whether a particular node has a state variable or not.

Investment Variable Types In all cases the capacity of the unit or connection or the maximum node state of a node is multuplied by the investment variable which may either be continuous, binary or integer. This is determined, for units, by setting the unit_investment_variable_type parameter accordingly. Similary, for connections and node storages where the connection_investment_variable_type and storage_investment_variable_type are specified.

Identiying Investment Candidate Units, Connections and Storages The parameter candidate_units represents the number of units of this type that may be invested-in. candidate_units determines the upper bound of the investment variable and setting this to a value greater than 0 identifies the unit as an investment candidate unit in the optimisation. If the unit_investment_variable_type is set to `:variable_type_integer`, the investment variable can be interpreted as the number of discrete units that may be invested in. However, if unit_investment_variable_type is `:variable_type_continuous` and the unit_capacity is set to unity, the investment decision variable can then be intpreted as the capacity of the unit rather than the number of units with candidate_units being the maximum capacity that can be invested in. Finally, we can invest in discrete blocks of capacity by setting unit_capacity to the size of the investment capacity blocks and have unit_investment_variable_type set to `:variable_type_integer` with candidate_units representing the maximum number of capacity blocks that may be invested in. The key points here are:

- The upper bound on the relevant flow variables are determined by the product of the investment variable and the unit_capacity or connection_capacity for connections or node_state_cap for storages.
- candidate_units sets the upper bound on the investment variable, candidate_connections for connections and candidate_storages for storages
- unit_investment_variable_type determines wheter the investment variable is integer, binary or continuous (connection_investment_variable_type for connections and storage_investment_variable_type for storages).

Investment Costs Investment costs are specified by setting the appropriate `*_investment\_cost` parameter. The investment cost for units are specified by setting the unit unit_investment_cost parameter. This is currently interpreted as the full cost over the investment period for the unit. See the section below on **investment temporal structure** for setting the investment period. If the investment period is 1 year, then the corresponding unit_investment_cost is the annualised investment cost. For connections and storages, the investment cost parameters are connection_investment_cost and storage_investment_cost, respectively.

Temporal and Stochastic Structure of Investment Decisions SpineOpt's flexible stochastic and temporal structure extend to investments where individual investment decisions can have their own temporal and stochastic structure indepedent of other investment decisions and other model variables. A global temporal resolution for all investment decisions can be defined by specifying the relationship model__default_investment_temporal_block. If a specific temporal resolution is required for specific investment decisions, then one can specify the following relationships: - unit__investment_temporal_block for unit - connection__investment_temporal_block for connection - node__investment_temporal_block for storages. Specifying any of the above relationships will override the corresponding model__default_investment_temporal_block.

Similarly, a global stochastic structure can be defined for all investment decisions by specifying the relationship model__default_investment_stochastic_structure. If a specific stochastic structure is required for specific investment decisions, then one can specifying the following relationships: - unit__investment_stochastic_structure for unit - connection__investment_stochastic_structure for connection - node__investment_stochastic_structure for storages Specifying any of the above relationships will override the corresponding model__default_investment_stochastic_structure.

Creating an Investment Candidate Unit Example

If we have model that is not currently set up for investments and we wish to create an investment candidate unit, we can take the following steps.

- Create the unit object with all the relationships and parameters necessary to describe its function.
- Ensure that the number_of_units parameter is set to zero so that the unit is unavailable unless invested-in

- Set the candidate_units parameter for the unit to 1 to specify that a maximum of 1 new unit of this type may be invested-in by the model.
- Set the unit_investment_variable_type to `unit_investment_variable_type_integer` to specify that this is a discrete unit investment decision.
- Specify the unit_investment_lifetime of the unit to, say, 1 year to specify that this is the minimum amount of time this new unit must be in existence after being invested-in.
- Specify the investment period for this unit's investment decision in one of two ways
    - Define a default investment period for all investment decisions in the model as follows:
        - create a temporal_block with the appropriate resolution (say 1 year)
        - link this to your [model]{@ref} object by creating the appropriate model__temporal_block relationship
        - set it as the default investment temporal block by setting model__default_investment_temporal_block
    - Or, define an investment period unique to this investment decision as follows:
        - creating a temporal_block with the appropriate resolution (say 1 year)
        - link this to your model object by creating the appropriate model__temporal_block relationship
        - specify this as the investment period for your unit's investment decision by setting the appropriate unit__investment_temporal_block relationship
- Similarly to the above, define the stochastic structure for the unit's investment decision by specifying either model__default_investment_stochastic_structure or unit__investment_stochastic_structure
- Specifying your unit's investment cost by setting the unit_investment_cost parameter. Since we have defined the investment period above as 1 year, this is therefore the unit's annualised investment cost.

Model Reference

Variables for investments

Variable Name	Indices	Description
`units_invested_available`	`unit`, s, t	The number of invested in `units` that are available at a given (s, t)
`units_invested`	`unit`, s, t	The point-in-time investment decision corresponding to the number of `units` invested in at (s,t)
`units_mothballed`	`unit`, s, t	"Instantaneous" decision variable to

		mothball a `unit`
`connections_invested_available`	`connection`, s, t	The number of invested-in `connections`s that are available at a given (s, t)
`connections_invested`	`connection`, s, t	The point-in-time investment decision corresponding to the number of `connections`s invested in at (s,t)
`connections_decommissioned`	`connection`, s, t	"Instantaneous" decision variable to decommission a `connection`
`storages_invested_available`	`node`, s, t	The number of invested-in storages that are available at a given (s, t)
`storages_invested`	`node`, s, t	The point-in-time investment decision corresponding to the number of storages invested in at (s,t)
`storages_decommissioned`	`node`, s, t	"instantaneous" decision variable to decommission a storage

Relationships for investments

Relationship Name	Related Object Class List	
`model__default_investment_temporal_block`	`model`, `temporal_block`	Default te unit__investm
`model__default_investment_stochastic_structure`	`model`, `stochastic_structure`	Default st unit__inve
`unit__investment_temporal_block`	`unit`, `temporal_block`	Set t model__c
`unit__investment_stochastic_structure`	`unit`, `stochastic_structure`	Set st model__defau

Parameters for investments

Parameter Name	Object Class List	Description
`candidate_units`	`unit`	The number of additional `units` of this type that can be invested in
`unit_investment_cost`	`unit`	The total overnight investment cost per candidate `unit` over the model horizon
`unit_investment_lifetime`	`unit`	The investment lifetime of the `unit` - once invested-in, a `unit` must exist for at least this amount of time
`unit_investment_variable_type`	`unit`	Whether the `units_invested_available` variable is continuous, integer or binary
`fix_units_invested`	`unit`	Fix the value of `units_invested`
`fix_units_invested_available`	`unit`	Fix the value of `connections_invested_available`
`candidate_connections`	`connection`	The number of additional `connections` of this type that can be invested in
`connection_investment_cost`	`connection`	The total overnight investment cost per candidate `connection` over the model horizon
`connection_investment_lifetime`	`connection`	The investment lifetime of the `connection` - once invested-in, a `connection` must exist for at least this amount of time
`connection_investment_variable_type`	`connection`	Whether the `connections_invested_available` variable is continuous, integer or binary
`fix_connections_invested`	`connection`	Fix the value of `connections_invested`

`fix_connections_invested_available`	`connection`	Fix the value of `connection_invested_available`
`candidate_storages`	`node`	The number of additional storages of this type that can be invested in at `node`
`storage_investment_cost`	`node`	The total overnight investment cost per candidate storage over the model horizon
`storage_investment_lifetime`	`node`	The investment lifetime of the storage - once invested-in, a storage must exist for at least this amount of time
`storage_investment_variable_type`	`node`	Whether the `storages_invested_available` variable is continuous, integer or binary
`fix_storages_invested`	`node`	Fix the value of `storages_invested`
`fix_storages_invested_available`	`node`	Fix the value of `storages_invested_available`

Related Model Files

Filename	Relative Path	Description
constraint*units*invested_available.jl	\constraints	constrains `units_invested_available` to be less than `candidate_units`
constraint*units*invested_transition.jl	\constraints	defines the relationship between `units_invested_available`, `units_invested` and `units_mothballed`. Analagous to `units_on`, `units_started` and `units_shutdown`
constraint*unit*lifetime.jl	\constraints	once a `unit` is invested-in, it must remain in existence for at least `unit_investment_lifetime` -

		analagous to `min_up_time`.
constraint*units*available.jl	\constraints	Enforces `units_available` is the sum of `number_of_units` and `units_invested_available`
constraint*connections*invested_available.jl	\constraints	constrains `connections_invested_available` to be less than `candidate_connections`
constraint*connections*invested_transition.jl	\constraints	defines the relationship between `connections_invested_available`, `connections_invested` and `connections_decommissioned`. Analagous to `units_on`, `units_started` and `units_shutdown`
constraint*connection*lifetime.jl	\constraints	once a `connection` is invested-in, it must remain in existence for at least `connection_investment_lifetime` - analagous to `min_up_time`.
constraint*storages*invested_available.jl	\constraints	constrains `storages_invested_available` to be less than `candidate_storages`
constraint*storages*invested_transition.jl	\constraints	defines the relationship between `storages_invested_available`, `storages_invested` and `storages_decommissioned`. Analagous to `units_on`, `units_started` and `units_shutdown`
constraint*storage*lifetime.jl	\constraints	once a storage is invested-in, it must remain in existence for at least `storage_investment_lifetime` - analagous to `min_up_time`.

« Ramping and Reserves

Unit Constraints »

Unit Constraints

Unit constraints allow the user to define arbitrary linear constraints involving most of the problem variables. This section describes this function and how to use it.

Key Unit Constraint Concepts

1. **The basic principle**: The basic steps involved in forming a unit constraint are:

- Creating a user constraint object: One creates a new unit_constraint object which will be used as a unique handle for the specific constraint and on which constraint-level parameters will be defined.
- Specify which variables are involved in the constraint: this generally involves creating a relationship involving the unit_constraint object. For example, specifying the relationship unit__from_node__unit_constraint specifies that the corresponding unit_flow variable is involved in the constraint. The table below contains a complete list of variables and the corresponding relationships to set.
- Specify the variable coefficients: this will generally involve specifying a parameter named `*_coefficient` on the relationship defined above to specify the coefficient on that particular variable in the constraint. For example, to define the coefficient on the unit_flow variable, one specifies the unit_flow_coefficient parameter on the approrpriate unit__from_node__unit_constraint relationship. The table below contains a complete list of variables and the corresponding coefficient parameters to set.
- Specify the right-hand-side constant term: The constraint should be formed in conventional form with all constant terms moved to the right-hand side. The right-hand-side constant term is specified by setting the right_hand_side unit_constraint parameter.
    - Specify the constraint sense: this is done by setting the constraint_sense unit_constraint parameter. The allowed values are `==`, `>=` and `<=`.
    - Coefficients can be defined on some parameters themselves. For example, one may specify a coefficient on a node's demand parameter. This is done by specifying the relationship node__unit_constraint and specifying the demand_coefficient parameter on that relationship

1. **Piecewise unit_flow coefficients**: As described in operating_points, specifying this parameter decomposes the unit_flow variable into a number of sub operating segment variables named unit_flow_op in the model and with an additional index, `i` for the operating segment. The intention of this functionality is to allow unit_flow coefficients to be defined individually per segment to define a piecewise linear function. To accomplish this, the steps are as described above with the exception that one must define operating_points on the appropriate unit__from_node or unit__to_node as an array type with the dimension corresponding to the number of operating points and then set the unit_flow_coefficient for the appropriate unit__from_node__unit_constraint

relationship, also as an array type with the same number of elements. Note that if operating points is defined as an array type with more than one elements, unit_flow_coefficient may be defined as either an array or non-array type. However, if operating_points is of non-array type, corresponding unit_flow_coefficients must also be of non-array types.

2. **Variables, relationships and coefficient guide for unit constraints** The table below provides guidance regarding what relationships and coefficients to set for various problem variables and parameters.

Problem variable / Parameter Name	Relationship	Parameter
`unit_flow` (direction=from_node)	unit__from_node__unit_constraint	unit_flow_coefficient (non-array type)
`unit_flow` (direction=to_node)	unit__to_node__unit_constraint	unit_flow_coefficient (non-array type)
`unit_flow_op` (direction=from_node)	unit__from_node__unit_constraint	unit_flow_coefficient (array type)
`unit_flow_op` (direction=to_node)	unit__to_node__unit_constraint	unit_flow_coefficient (array type)
`connection_flow` (direction=from_node)	connection__from_node__unit_constraint	connection_flow_coefficient
`connection_flow` (direction=to_node)	connection__to_node__unit_constraint	connection_flow_coefficient
`node_state`	node__unit_constraint	node_state_coefficient
`demand`	node__unit_constraint	demand_coefficient

« Investment Optimization

Decomposition »

Decomposition

Decomposition approaches take advantage of certain problem structures to separate them into multiple related problems which are each more easily solved. Decomposition also allows us to do the inverse, which is to combine independent problems into a single problem, where each can be solved separately but with communication between them (e.g. investments and operations problems)

Decomposition thus allows us to do a number of things

- Solve larger problems which are otherwise intractable
- Include more detail in problems which otherwise need to be simplified
- Combine related problems (e.g. investments/operations) in a more scientific way (rather than ad-hoc).
- Employ parallel computing methods to solve multiple problems simultaneously.

High-level Decomposition Algorithm

The high-level algorithm is described below. For a more detailed description please see Benders decomposition

- Model initialisation (preprocess*data*structure, generate temporal structures etc.)
- For each benders_iteration
  - Solve master problem
  - Process master-problem solution:
    - set `units_invested_bi(unit=u, benders_iteration=bi)` equal to a timeseries representing the investment variables solution from the master problem
  - Rewind and update operations problem
  - Solve operations problem loop
  - Process operations sub-problem
    - set `units_available_mv(unit=u, benders_iteration=bi)` equal to a timeseries representing the marginal value of the units_on bound constraint
  - Test for convergence
  - Update master problem
    - Add Benders cuts constraints
  - Next benders iteration

Duals calculation for decomposition

The `optimize_model!()` function has been updated to optionally include an additional step for the calculation of duals. The dual solution to a MIP problem is not well defined. The standard approach to obtaining marginal values from a MIP model is to relax the integer variables, fix them to their last solution value and re-solve the problem as an LP. This is the standard approach in energy system modelling to obtain energy prices. However, although this is the standard approach, it does need to be used with caution (see here for example). The main hazard associated with inferring duals in this way is that the impact on costs of an investment may be overstated. However, since these duals are used in Benders decomposition to obtain a lower bound on costs (i.e. the maximum potential value from an investment), this is ok and can be "corrected" in the next iteration. And finally, the benders gap will tell us how close our decomposed problem is to the optimal global solution.

This additional relaxed LP solve is done as follows:

- `add_variable!()` stores the list of integer and binary variables in `m.ext[:integer_variables]`
- the `fix_value` for integer variables is set to the last MIP solution value
- the integer constraints on the integer variables are `unset()`
- A final LP is solved
- required dual values are saved
- integer constraints on integer variables are `set()`

This final fixed LP solve is trigged by specifying `calculate_duals=true` in the call to `optimize_model!()`

Reporting dual values:

To report the dual of a constraint, one can add an output item with the corresponding constraint name (e.g. `constraint_nodal_balance`) and add that to a report. This will cause the corresponding constraint's relaxed problem marginal value will be reported in the output DB. When adding a constraint name as an output we need to preface the actual constraint name with `constraint_` to avoid ambiguity with variable names (e.g. `units_available`). So to report the marginal value of `units_available` we add an output object called `constraint_units_available`.

To report the `reduced_cost()` for a variable which is the marginal value of the associated active bound or fix constraints on that variable, one can add an output object with the variable name prepended by `bound_`. So, to report the units*on reduced*cost value, one would create an output item called `bound_units_on`. If added to a report, this will cause the reduced cost of units*on in the final fixed LP to be written to the output db. Finally, if any constraint duals or reduced*cost values are requested via a report, calculate_duals is set to true and the final fixed LP solve is triggered.

Using Decomposition

The decomposition framework creates a master problem where the investment variables are optimised. The decomposition framework is invoked when a model object with the parameter model_type set to

`:spineopt_operations` is found and a second model object with `model_type` set to `:spineopt_master`. Once these conditions are met, all investment decisions in the model are automatically decomposed and optimised in the master problem. This behaviour may change in the future to allow some investment decisions to be optimised in the operations problem and some optimised in the master problem as desired.

Steps to involke decomposition in an investments problem Assuming one has set up a conventional investments problem as described in Investment Optimization the following additional steps are required to utilise the decomposition framework:

- Create a new model object to representent the benders master problem
- Set the model_type parameter for the master problem model to `spineopt_master`.
- Set the model_type parameter for the existing conventional, operations problem model to `spineopt_operations`.
- Specify the master problem model parameter, `max_gap` - This determines the master problem convergence criterion for the relative benders gap. A value of 0.05 will represent a relative benders gap of 5%.
- Specify the master problem model parameter `max_iterations` - This determines the master problem convergence criterion for the number of iterations. A value of 10 could be appropriate but this is highly dependent on the size and nature of the problem
- Specify appropriate `model_report` relationships to determine which reports are written for which model

---

« Unit Constraints PTDF-Based Powerflow »

Power transfer distribution factors (PTDF) based DC power flow

There are two main methodologies for directly including DC powerflow in unit commitment/energy system models. One method is to directly include the bus voltage angles as variables in the model. This method is described in Nodal lossless DC Powerflow.

Here we discuss the method of using power transfer distribution factors (PTDF) for DC power flow and line outage distribution factors (lodf) for security constrained unit commitment.

Key concepts

1. **ptdf**: The power transfer distribution factors are a property of the network reactances and their derivation may be found here. `ptdf(n, c)` represents the fraction of an injection at node n that will flow on connection c. The flow on connection c is then the sum over all nodes of `ptdf(n, c)*net_injection(c)`. The advantage of this method is that it introduces no additional variables into the problem and instead, introduces only one constraint for each connection whose flow we are interested in monitoring.
2. **lodf**: Line outage distribution factors are a function of the network ptdfs and their derivation is also found here. `lodf(c_contingency, c_monitored)` represents the fraction of the pre-contingency flow on connection `c_contingency` that will flow on `c_monitored` if `c_contingency` is disconnected. Therefore, the post contingency flow on connection `c_monitored` is the `pre_contingency flow` plus `lodf(c_contingency, c_monitored)\*pre_contingency_flow(c_contingency))`. Therefore, consideration of N contingencies on M monitored lines introduces N x M constraints into the model. Usually one wishes to contain this number and methods are given below to achieve this.
3. **Defining your network** To identify the network for which ptdfs, lodfs and connection_flows will be calculated according to the ptdf method, one does the following:

    - Create node objects for each bus in the model.
    - Create connection objects representing each line of the network: For each connection specify the connection_reactance parameter and the connection_type parameter. Setting connection_type=`connection_type_lossless_bidirectional` simplifies the amount of data that needs to be specified for an eletrical network. See connection_type for more details
    - Set the connection__to_node and connection__from_node relationships to define the topology of each connection along with the connection_capacity parameter on one or both of these relationships.
    - Set the connection_emergency_capacity parameter to define the post contingency rating if lodf-based N-1 security constraints are to be included

- Create a commodity object and node__commodity relationships for all the nodes that comprise the electrical network for which PTDFs are to be calculated.
- Specify the commodity_physics parameter for the commodity to `:commodity_physics_ptdf` if ptdf-based DC load flow is desired with no N-1 security constraints or to `:commodity_physics_lodf` if it is desired to include lodf-based N-1 security constraints
- To identify the reference bus(node) specify the node_opf_type parameter for the appropriate node with the value `node_opf_type_reference`.

4. **Controlling problem size**

- The lines to be monitored are specified by setting the connection_monitored property for each connection for which a flow constraint is to be generated
- The contingencies to be considered are specified by setting the connection_contingency property for the appropriate connections. For N contingencies and M monitored lines, N x M constraints will be generated.
- If the `lodf(c_contingency, c_monitored)` is very small, it means the outage of `c_contingency` has a small impact on the flow on `c_monitored`and there is little point in including this constraint in the model. This can be achieved by setting the `commodity_lodf_tolerance` commodity parameter. Contingency / Monotired line combinations with lodfs below this value will be ignored, reducing the size of the model.
- If `ptdf(n, c)` is very small, it means an injection at n has a small impact on the flow on c and there is little point in considering it. This can be achieved by setting the `commodity_ptdf_threshold` commodity parameter. Node / Monotired line combinations with ptdfs below this value will be ignored, reducing the number of coefficients in the model.

« Decomposition

Pressure driven gas transfer »

Pressure driven gas transfer

The generic formulation of SpineOpt is based on a trade based model. However, network physics can be different depending on the traded commodity. This chapter specifically addresses the use of pressure driven gas transfer models and enabling linepack flexibility in SpineOpt. To this date, investments in pressure driven pipelines are not yet supported within SpineOpt. The use of multiple feed-in nodes, e.g. to represent multiple commodity flows through a pipeline is not yet supported.

For the representation of pressure driven gas transfer, we use the MILP formulation, as described in Schwele - Coordination of Power and Natural Gas Systems: Convexification Approaches for Linepack Modeling. Here, the non-linearities associated with the Weymouth equation are convexified through an outer approximation of the Weymouth equation through fixed pressure points.

Key concept

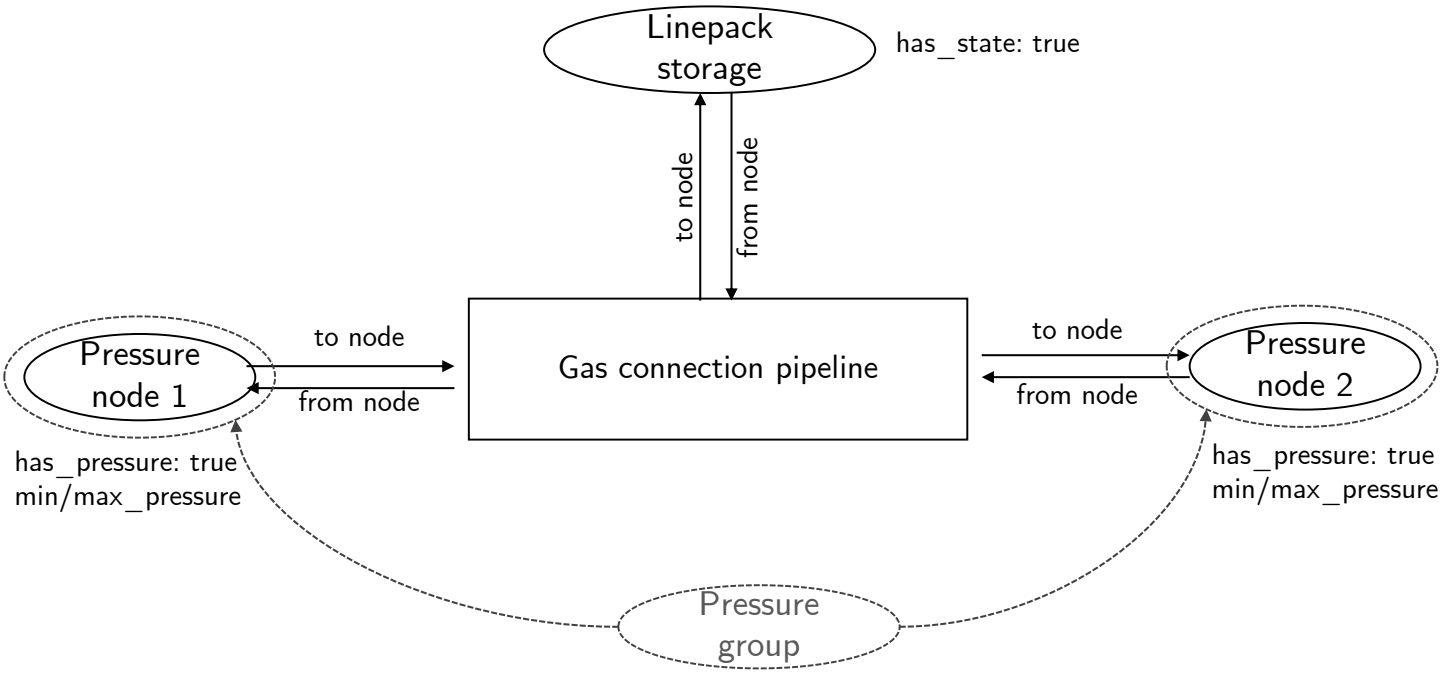
Here, we briefly describe the key objects and relationships required to model pressure driven gas transfers in SpineOpt.

1. **connection**: A connection represents the gas pipeline being modelled. Usually the direction of flow is not known a priory. To ensure that the flow through the gas pipeline is unidirectional, the parameter has_binary_gas_flow needs to be set to `true`.
2. **node**: Nodes with different characteristics are used for the representation of pressure driven gas transfer.
    - For each connection, there will be two nodes representing the start and end point of the pipeline. Associated with these nodes are the following parameters: the has_pressure parameter, which needs to be set to true, in order to create the variable node_pressure; the max_node_pressure and min_node_pressure to constrain the pressure variable.
    - To leverage linepack flexibility, a third node is introduced representing the linepack storage of the pipeline. To trigger the storage linepack and hence, node_state variables, the has_state parameter needs to be set to `true`.
3. **connection__to_node** and **connection__from_node** To enable flows through the pipeline and into the linepack storage, each node has to have both these relationships in common with the connection pipeline. These relationships will trigger the generation of connection_flow variables in all possible directions.
4. **connection__node__node** This relationship is key to the pressure driven gas transfer, holding the information about the pipeline characteristics and bringing the elements into interaction.
    - The parameter connection_linepack_constant holds the linepack constant and triggers the generation of the line pack storage constraint. Note that the first node should be the linepack

storage node, while the second node should be a node_group of both, the start and the end node of the pipeline.

- The linearization of the Weymouth equation through outer approximation relies on the use of fixed pressure points. For this purpose, the two parameters fixed_pressure_constant_1 and fixed_pressure_constant_0 hold the fixed pressure constants and trigger the generation of the constraint_fix_node_pressure_point. The constraint introduces the relationship between pressure and gas flows. Note, that the pressure constants should be entered in a way, that the first node represents the origin node, the second node the destination node. Each connection should have a connection__node__node to each combination of its start and end nodes (and associated parameters). (See Schwele - Coordination of Power and Natural Gas Systems: Convexification Approaches for Linepack Modeling)
- By default, pipelines are considered to be passive. However, a compression station between two pipeline pressure nodes can be represented by defining a compression_factor. The relationship should be defined in such a manner, that the first node represents the sending node, the second node represents the receiving node, which pressure is equal or smaller to the pressure at the sending node times the compression factor.
- Lastly, to ensure the balance between incoming/outgoing flows and flows into the linepack, the ratio between the flows need to be fixed. The average incoming flows of the node group (of the pressure start and end nodes) have to equal the flows into the linepack storage, and vice versa. Therefore, the fix_ratio_out_in_connection_flow needs to be set to a value (typically 1) for the (pressure group, linepack storage) node pair, and for the (linepack storage, pressure group) node pair.

A gas pipeline and its connected nodes are illustrated below. A complete mathematical formulation can be found here.



« PTDF-Based Powerflow
Lossless nodal DC power flows »

Lossless nodal DC power flows

Currently, there are two different methods to represent lossless DC power flows. In the following the implementation of the nodal model is presented, based of node voltage angles.

Key concepts

In the following, it is described how to set up a connection in order to represent a nodal lossless DC power flow network. Therefore, key object - and relationship classes as well as parameters are introduced.

1. **connection**: A connection represents the electricity line being modelled. A physical property of a connection is its connection_reactance, which is defined on the connection object. Furthermore, if the reactance is given in a p.u. different from the standard unit used (e.g. p.u. = 100MVA), the parameter connection_reactance_base can be used to perform this conversion.
2. **node**: In a lossless DC power flow model, nodes correspond to buses. To use voltage angles for the representation of a lossless DC model, the has_voltage_angle needs to be `true` for these nodes (which will trigger the generation of the node_voltage_angle variable). Limits on the voltage angle can be enforced through the max_voltage_angle and min_voltage_angle parameters. The reference node of the system should have a voltage angle equal to zero, assigned through the parameter fix_node_voltage_angle.
3. **connection__to_node** and **connection__from_node** : These relationships need to be introduced between the connection and each node, in order to allow power flows (i.e. connection_flow). Furthermore, a capacity limit on the connection line can be introduced on these relationships through the parameter connection_capacity.
4. **connection__node__node**: To ensure energy conservation across the power line, a fixed ratio between incoming and outgoing flows should be given. The fix_ratio_out_in_connection_flow parameter enforces a fixed ratio between outgoing flows (i.e. to_node) and incoming flows (i.e. from_node). This parameter should be defined for both flow direction.

The mathematical formulation of the lossless DC power flow model using voltage angles is fully described here.

Representative days with seasonal storages

In order to reduce computational times, representative periods are often used in optimization models. However, this often limits the ability to properly account for seasonal storages.

In SpineOpt, we provide functionality to use representative days with seasonal storages in combination with the package **SpinePeriods.jl**.

General idea

The general idea is to mimick the seasonal effects throughout a non-representative period, e.g. a year of optimization, by introducing a specific sequence of the representative periods. Taking the example of one year to be optimized with representative days and seasonal storages, **SpinePeriods.jl** provides a mapping of each day of the year to its corresponding representative day. This information is stored in the mapping parameter representative_periods_mapping and is defined on the temporal_block for the whole year. The representative_periods_mapping parameter is a timeseries, pointing the beginning of each day to its corresponding representative day temporal_block, which can also be automatically be generated through **SpinePeriods.jl**.

In SpineOpt, this is interpreted in the following way:

- All operational variables, with the exception of the node_state variable, are created for each representative period. For each non-representative period, the variables are mapped to their corresponding variable of the representative periods according to the representative_periods_mapping parameter.
- Only the node_state variables and all investment variables are created for both, representative and non-representative period (of course, depending on the existance of relationships to temporal_blocks).

Usage of representative days and seasonal storages for investment problems

To make use of representative days with seasonal storages concept, multiple temporal_block objects need to be created and connected to the system components, holding information about the resolutions in different parts of the model that come into play. As described in the section Temporal Framework, every temporal block needs to be connected to a model object.

- **temporal_block for investments**: In order to define the resolution of the investment decisions, a temporal_block relecting the frequency of investment decisions should be introduced. For yearly investment, the resolution of this temporal block would be equal to `1Y`. In order to link nodes, units

or connections to this investment resolution, the node__investment_temporal_block, unit__investment_temporal_block, or connection__investment_temporal_block relationships need to be defined, respectively. For more details on investments, see also section Investment Optimization

- **temporal_block for representative days**: For each representative day, one temporal_block needs to be created, indicating the block_start and block_end of the representative day. The use of disconnected periods is also described in the section Disconnected time periods. The resolution of the representative days corresponds to the resolution of the operational variables, e.g. `1h`. In order to associate operational variables with the representative periods, node__temporal_block and units_on__temporal_block relationships need to be created. For convenience, it is also possible to create a group of all representative temporal_blocks and link this group to these relationships. Note that, when using **SpinePeriods.jl**, the representative temporal blocks are auto-generated.
- **temporal_block for non-representative days**: To introduce node_state variables for the entire operational period, a temporal block overarching the entire horizon is created. Note that currently, this temporal block needs to have the same resolution as the representative days, e.g. `1h`. In order to associate operational variables with the representative periods, node__temporal_block and units_on__temporal_block relationships need to be created. Note that, as described above, the non-representative variables, will be mapped to their corresponding representative days. To manually introduce the mapping between non-representative and representative periods, instead of using the recommended SpinePeriods.jl, the user must define the mapping parameter representative_periods_mapping by hand, consisting of `DateTime` indices (indicating the start of each non-representative period, e.g. for a daily mapping `2021-01-01T00:00:00`, `2021-01-02T00:00:00` etc.) and the name of the corresponding representative `temporal_block` as a value.